UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Trabajo Final de Grado

Grado en Ingeniería Informática (GEI)
Especialidad Ingeniería de Computadores

Viabilidad Del Uso De Contenedores En Entornos HPC

Trabajo realizado en el Barcelona Supercomputing Center (BSC)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1 de julio de 2019

Autor: Oleksandr Rudyy

Director: Raül Sirvent Pardell (BSC)

Codirectora: Marta Garcia Gasulla (BSC)

Ponente: Jordi Guitart Fernández
(Dpto. Arquitectura de Computadors)

Abstract

Container technologies have changed the way services and applications are deployed in the cloud. However, their adoption in High-Performance Computing (HPC) centers is still under discussion: on one hand, the ease in portability is very attractive; on the other, it is not clear whether containers can fulfill HPC performance and security requirements. Since very little evaluation of large production HPC codes running in containers is available, in this work we provide a comparative study using a production scientific application (Alya) besides synthetic benchmarks (BT-MZ and HPCG) to validate the results. We analyze the productivity advantages of adopting containers for large HPC codes and we quantify the performance overhead induced by the use of three different container implementations (Docker, Singularity, and Shifter) comparing it to native executions. Given the results of these tests, we selected Singularity as the best candidate based on performance and portability. We present a portability and performance evaluation of containers across three different HPC architectures (Intel Skylake, IBM Power9, and Armv8-a) and show scalability results of Alya using Singularity up to 256 computational nodes (up to 12k cores) of the BSC supercomputer MareNostrum4.

Resumen

Las tecnologías de contenedores han cambiado cómo los servicios y aplicaciones son desplegados en la nube. Sin embargo, su adopción en centros de *High-Performance Computing* (HPC) aún está bajo discusión: por un lado, resulta muy atractiva la portabilidad de las aplicaciones; por el otro, no está claro si los contenedores satisfacen los requisitos HPC de rendimiento y seguridad. Dado que existen pocas evaluaciones de códigos HPC significantes ejecutándose en contenedores, nosotros aportamos con este trabajo un estudio comparativo usando una aplicación científica en producción (Alya), además de *benchmarks* sintéticos (BT-MZ y HPCG) para validar los resultados. En este trabajo, analizamos las ventajas que ofrece la adopción de los contenedores con un código HPC relevante y cuantificamos el sobrecoste en rendimiento inducido por el uso de tres implementaciones diferentes de contenedores (Docker, Singularity y Shifter) comparándolo con ejecuciones nativas. Dados los resultados de estos experimentos, hemos elegido Singularity como el mejor candidato basándonos en su rendimiento y portabilidad. Los experimentos incluyen una evaluación del rendimiento y portabilidad de los contenedores a través de tres sistemas HPC distintos (Intel Skylake, IBM Power9 y Armv8-a) y los resultados de la escalabilidad de Alya usando Singularity hasta 256 nodos computacionales (hasta 12 mil cores) del supercomputador del BSC MareNostrum4.

Resum

Les tecnologies de contenidors han canviat com els serveis i les aplicacions són desplegades en el núvol. Malgrat això, la seva adopció en centres de *High-Performance Computing* (HPC) encara està sota discussió: per una banda, resulta molt atractiva la portabilitat de les aplicacions; per l'altra, no està clar si els contenidors satisfan els requisits HPC de rendiment i seguretat. Donat que existeixen poques avaluacions de codis HPC significants executant-se en contenidors, nosaltres aportem amb aquesta feina un estudi comparatiu utilitzant una aplicació científica en producció (Alya) a més de *benchmarks* sintètics (BT-MZ i HCPG) per validar els resultats. En aquest treball analitzem els avantatges que ofereix l'adopció dels contenidors amb un codi HPC rellevant i quantifiquem el sobrecost en rendiment imbuït per l'ús de tres implementacions diferents de contenidors (Docker, Singularity i Shifter) comparant-los amb execucions natives. D'acord amb els resultats, hem escollit Singularity com el millor candidat basant-nos en el seu rendiment i portabilitat. Als experiments presentem una avaluació del rendiment i portabilitat dels contenidors a través de 3 sistemes HPC diferents (Intel Skylake, IBM Power9 i Armv8-a) i mostrem els resultats de l'escalabilitat d'Alya fent servir Singularity fins a 256 nodes computacionals (fins a 12 mil cores) del supercomputador del BSC MareNostrum4.

Índice General

Índice de Figuras

Índice de Tablas

Glosario

ABI Application Binary Interface. 16

ALE Arbitrary Lagrangian-Eulerian. 21

BSC Barcelona Supercomputing Center. 3, 9, 51–53, 59, 63, 66–68

BT-MZ Block Tri-diagonal solver Multi-Zone. 19, 21, 23, 24, 26, 28, 37, 39, 41, 42, 53, 55, 64

CASE Computer Applications in Science and Engineering. 53

CFD Computational Fluid Dynamics. 21, 23, 42, 44, 47, 64

CLI Command Line Interface. 16

CSM Computational Solid Mechanics. 21

E/S Entrada/Salida. 7

EDP Ecuaciones en Derivadas Parciales. 24

FSI Fluid-Structure Interaction. 21, 27, 37

GPU Graphics Processing Unit. 8, 14–16, 72

HPC High-Performance Computing. 3, 4, 8, 9, 11–16, 20, 22, 25, 26, 31, 33, 42–44, 48, 49, 67–69, 71–73

HPCG High-Performance Coarse Gradient. 19, 21, 24, 26, 28, 37, 41, 42, 53, 59, 64

IPC Inter Process Communication. 6

IPDPS International Parallel and Distributed Processing Symposium. 73

ISA Instruction Set Architecture. 3, 4

LXC Linux Containers. 13

MNT Mount. 6

MPI Message Passing Interface. 14–16, 19–23, 27, 28, 33–39, 41–47, 49, 50, 71, 72

NASA National Aeronautics and Space Administration. 23

NERSC National Energy Research Scientific Computing Center. 11, 16

NET Network. 6

NPB NASA Advanced Supercomputing Parallel Benchmarks. 23

OSI Open Systems Interconnection (modelo de interconexión de sistemas abiertos). 35

PID Process Identifier. 6, 7

PMI Process Management Interface. 15

PPP Proyecto Puesto en Producción. VIII, 67

rkt rocket. 13

SD Standard Deviation. 36

SUID Set User ID. 15, 16

TFG Trabajo de Fin de Grado. 3, 11, 25, 54

TI Tecnologías de la Información. 71

UEABS Unified European Applications Benchmark Suite. 21

USR User. 6

UTS Unix Timesharing System. 6

WP12 - JRA1 Work Package 12 - Joint Research Activity. 3

Agradecimientos

Siento infinita gratitud hacia mi director y codirectora Raül y Marta, por todo su apoyo desde el picosegundo cero y sus esfuerzos invertidos en mi para enseñarme a realizar un estudio académico. Gracias a Filippo, quien sin apenas conocerme, me ha ayudado en todo lo que ha podido y mucho más. También quiero agradecer a Alfonso y Mariano del departamento de CASE del BSC por su interés en colaborar y buen humor. Gracias a Jordi Guitart por aceptar el cargo de ser el ponente de este trabajo y aportar su experiencia. Y gracias al departamento de Operaciones del BSC, ya que siempre se ha mostrado dispuesto a atender mis peticiones en todo lo que ha hecho falta. He contraído una deuda impagable con estas personas, producto de sus virtudes como profesionales y bondad como personas.

Fuera del marco exclusivo del TFG, también siento cariño hacia mi grupo de amigos y compañeros de grado del *Soviet Supremo Esmeralda*: Enroc, Megadri, GFountains, DjCubiCubi y Scubidovi, con quienes he podido compartir los mejores y peores momentos de la carrera. Por último, pero no por ello menos importante, gracias a mis compañeros de trabajo Mario, Joel y Marc, camaradas de oficina siempre dispuestos a echar una mano y hacer piña.

Este trabajo está parcialmente financiado por el programa de investigación e innovación 2020 *European Union's Horizon* a través del proyecto HPC-EUROPA3 (INFRAIA-2016-1-730897), el proyecto CompBioMed (acuerdo de subvención 675451 H2020-EU1.4.1.3), el proyecto Mont-Blanc 3 (acuerdo de subvención 671797), el Gobierno de España a través del programa Severo Ochoa (SEV-2015-0493), por el Ministerio Español de Ciencia y Tecnología mediante el proyecto TIN2015-65316-P y la Generalitat de Catalunya (contrato 2017-SGR-1414). También queremos agradecer a Lenovo por dejarnos usar su clúster Lenox para los experimentos de este trabajo.

Capítulo 1

Introducción

Este Trabajo de Fin de Grado (TFG) está realizado dentro de la modalidad de proyectos realizados en empresa en el Barcelona Supercomputing Center (BSC). Dentro del BSC, este proyecto surge de la participación en las tareas del Work Package 12 - Joint Research Activity (WP12 - JRA1) Container-as-a-service for HPC, del proyecto HPC-Europa3 [1].

En las siguientes secciones de este capítulo se sitúa al lector sobre el contexto del proyecto. Se explica el escenario actual donde se desarrolla, se exponen concisamente los detalles técnicos, se muestra el problema que se quiere abordar y finalmente los agentes implicados. Los siguientes capítulos tratan directamente el trabajo desarrollado. Se empieza explorando el estado del arte, luego se describe el hardware y software utilizado dando paso a la metodología del proyecto y los 3 experimentos conducidos: una comparativa entre implementaciones de contenedores, un estudio de portabilidad entre arquitecturas y un test de escalabilidad. Al final del documento, se discute la planificación, costes y sostenibilidad del TFG. El último capítulo está dedicado a las conclusiones finales del proyecto.

1.1. Contexto

Uno de los retos a los que se enfrenta habitualmente el sector del *High-Performance Computing* (HPC) no sólo es adaptar sus aplicaciones a los últimos avances científico-tecnológicos, sino también adaptarse a la inmensa heterogeneidad de sistemas informáticos. Los centros de datos y centros HPC están deviniendo cada vez más y más complejos, tanto des del punto de vista software como hardware. Las capas software necesarias para gestionar todo un clúster de computadores requieren de un despliegue lento, a prueba de errores y muchas veces personalizado. Además, la moda actual del hardware especializado incita a usar conjuntos de instrucciones o *Instruction Set Architectures* (ISA) distintos del más común x86 de Intel, dificultando el trabajo a administradores de sistemas y usuarios. Por un lado, se requiere obtener el máximo rendimiento posible del hardware. Por el otro lado, existe el compromiso de sacrificar rendimiento para mejor la portabilidad de las aplicaciones.

A parte de la heterogeneidad de sistemas, las características innatas de las aplicaciones

[1] Página web de HPC-Europa3: http://www.hpc-europa.eu/research_activities

HPC también influyen en su portabilidad. Es normal en ámbitos de investigación manejar programas con centenares de miles de líneas de código, y que a la vez estén enlazados con otro software o entornos de ejecución. En consecuencia, las tareas de instalación, actualización y adaptación de software y aplicaciones HPC en supercomputadores resultan complejas. Tampoco hay garantías de que una aplicación acabe siendo viable para ejecutar en una máquina nueva, ya sea por divergencias con las versiones de librerías que contiene el nuevo sistema o restricciones hardware. En definitiva, **la comunidad HPC carece de herramientas que abstraigan los requisitos software de sus aplicaciones del entorno de ejecución final.**

1.2. Tecnologías de Virtualización

Una potencial solución para todos los problemas expuestos es la virtualización. Mediante la técnica de la virtualización es posible almacenar en un fichero llamado *imagen* todo el software necesario, ya sea una aplicación, sistema operativo, controladores de dispositivos, librerías externas, etc. Luego, con esta imagen se puede desplegar un sistema virtual (una máquina huésped) dentro de un computador (la máquina anfitrión) de manera rápida, totalmente portable y compatible[2]. En la Figura 1.1 se ilustra el funcionamiento de la tecnología de virtualización, donde se muestra cómo un sistema informático se puede desplegar sistemáticamente en varios ordenadores mediante una imagen.

Los avances en las tecnologías de virtualización no sólo han facilitado su accesibilidad, sino también su usabilidad. Con herramientas como VMware [1], VirtualBox [2] o QEMU [3], [4] la virtualización está al alcance de todos. Si a su accesibilidad y simplicidad le sumamos la abundancia y bajo coste del hardware actual, vemos cómo las plataformas de virtualización son entornos ideales sobre los que desplegar servicios en la nube tal como los conocemos hoy en día.

Desgraciadamente, las técnicas de virtualización han quedado históricamente fuera del alcance del sector HPC. La virtualización convencional añade capas software al sistema que ralentizan considerablemente el rendimiento de las aplicaciones. Como por definición el sector HPC busca el mínimo tiempo de ejecución de sus aplicaciones, la virtualización quedó descartada por no cumplir sus requisitos de rendimiento.

He aquí cuando aparece por fin el objeto de estudio de este trabajo. En la última década la tecnología de virtualización ha concebido lo que técnicamente se llama "virtualización a nivel de sistema operativo" o popularmente "contenedores". Los contenedores, a diferencia de las técnicas de virtualización convencionales, usan el kernel Linux de su anfitrión en lugar de uno propio. En otras palabras, los contenedores se ejecutan como un proceso normal en el anfitrión, lo cual permite acceder al hardware sin sobrecostes añadidos. Esta característica convierte a los contenedores en un proceso ideal para "contener" todo el entorno software de una aplicación y automatizar su ejecución evitando todo el proceso de instalación. En la Figura 1.2 se muestra con un esquema la diferencia entre las tecnologías de virtualización convencionales (virtualización completa o *Full Stack Virtualization*) y los contenedores. Básicamente, los contenedores ofrecen al sistema informático virtual acceso directo al hardware del anfitrión, mientras que las demás tecnologías requieren de una capa software entre medio (el *hypervisor*). Por el contrario, los contenedores no pueden virtualizar el hardware (ISA, controladores, dispositivos varios,

[2]Compatible siempre que el anfitrión tenga disponible la tecnología de virtualización usada.

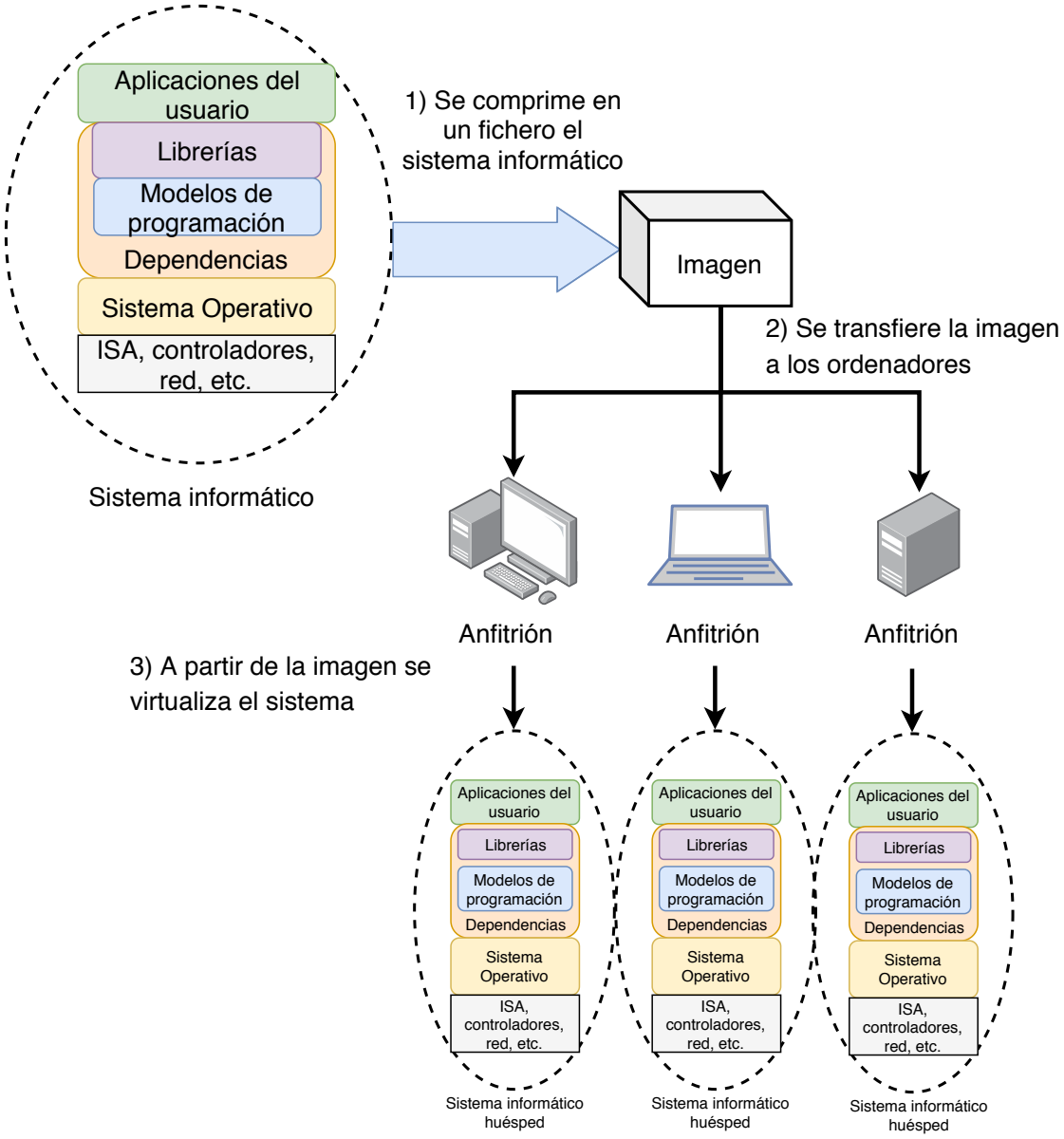


Figura 1.1: Ejemplo de uso de la virtualización.

etc.), sino solamente abstraer el software. A continuación se precisa el funcionamiento de los contenedores y sus características.

1.2.1. Contenedores

En esencia, los contenedores son un tipo de de virtualización ligera que prescinden del uso de *hypervisors* o monitores de máquinas virtuales. Para ello, los contenedores explotan una funcionalidad del kernel de Linux que permite gestionar los recursos disponibles de cada proceso: los espacios de nombres o `namespace`. Mediante la gestión de los `namespace` se es capaz de tener procesos con distintas jerarquías del sistema de ficheros, distintos dispositivos accesibles (redes, tarjetas gráficas, discos, etc.), memoria compartida aislada o nombre del equipo (*hostna-*

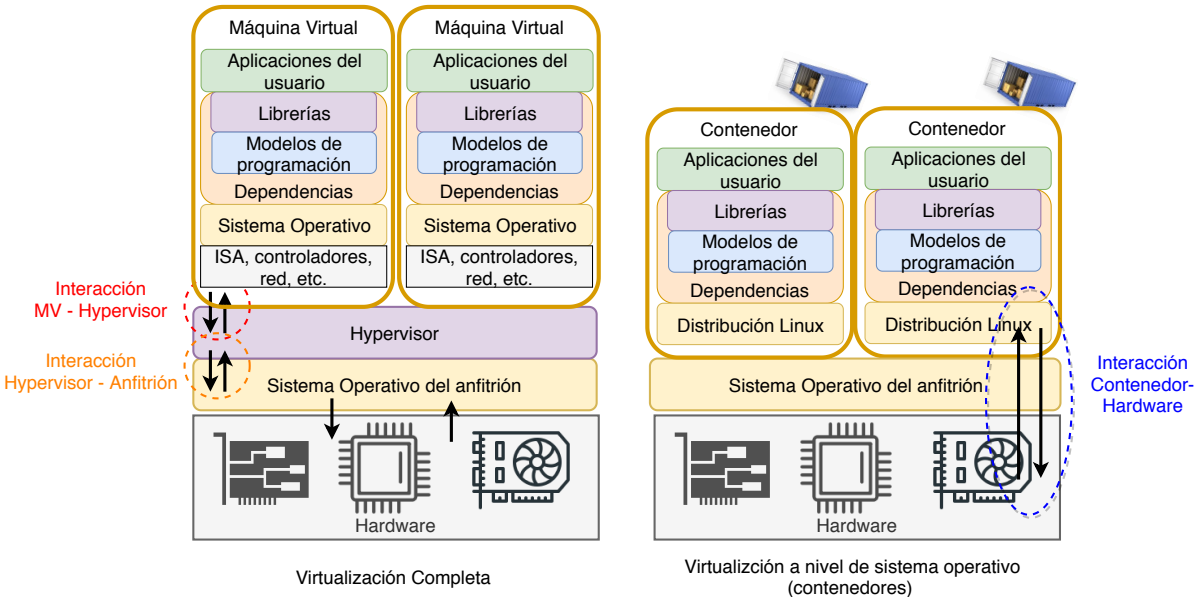


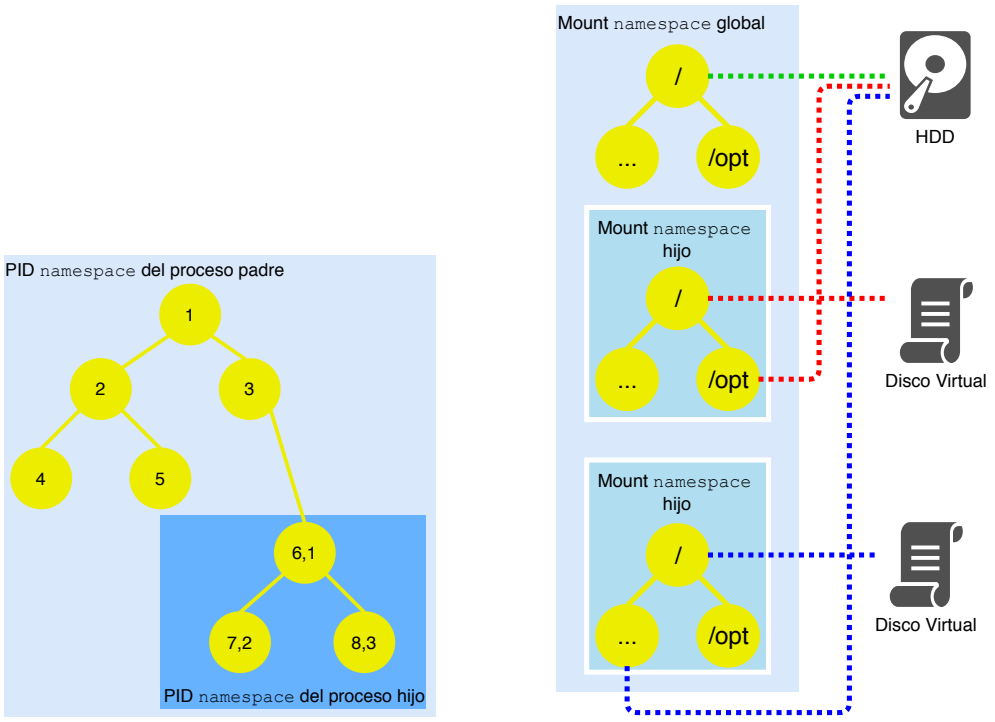
Figura 1.2: Funcionamiento de la virtualización completa (izquierda) y los contenedores (derecha).

me) y nombre del dominio al que pertenece éste (*domain name*) diferente de otros procesos. Todo esto se logra mediante la gestión de los 6 `namespace` implementados actualmente:

- *Identificador del proceso (PID)* - Ofrece un conjunto independiente de identificadores de proceso de otros `namespace`.
- *Red (NET)* - Virtualiza la red del anfitrión. Cada espacio de nombres de red puede contener conjuntos de direcciones IP privadas, tablas de enrutamiento, dispositivos de red, etc.
- *Comunicación entre procesos (IPC)* - Aísla los métodos de comunicación System V y POSIX. Es decir, permite definir dominios de memoria compartida disjuntos.
- *Mount (MNT)* - Permite tener distintas vistas de la jerarquía del sistema de ficheros.
- *Usuario (USR)* - Gestiona los identificadores de usuario y grupo y permite modificar sus privilegios.
- *Unix Timesharing System (UTS)* - Habilita que los procesos tengan un *hostname* (nombre del anfitrión) y NIS (*Network Information System*) diferentes.

Es decir, los espacios de nombres permiten abstraer los recursos del sistema en distintos conjuntos, y éstos pueden tener elementos compartidos o disjuntos. A saber, en la Figura 1.3 se muestra un ejemplo en el caso del PID y Mount `namespace`: en la Figura 1.3a se nota como el hijo no puede ver el conjunto de PIDs del padre; en la Figura 1.3b se muestra que cada `namespace` de Mount puede definir su propia jerarquía del sistema de ficheros usando todo tipo de dispositivos.

Además de los `namespace`, los contenedores también tienen la posibilidad de utilizar los `cgroups` (control de grupos) para gestionar mejor el uso de los recursos hardware (CPU, memoria, disco, entrada/salida, red). Los `cgroups` es otra funcionalidad del kernel de Linux que per-



(a) Representación del PID `namespace`. (b) Representación del Mount `namespace`.

Figura 1.3: Ejemplos de uso de los `namespace`.

mite definir y monitorizar exactamente la cantidad de recursos que pueden usar los procesos, a diferencia de los `namespace` los cuales sólo definen a qué recursos se tiene acceso. Exactamente ofrecen las siguientes funcionalidades.

- Limitación de recursos: a los procesos se le puede limitar la cantidad de memoria, CPU o periféricos que pueden usar.
- Prioridad: define la porción de CPU o ancho de banda de E/S (entrada/salida) disponible para cada grupo.
- Contabilidad: mide la cantidad de recursos usados por un grupo.
- Control: controla el flujo de ejecución de los procesos (los congela, suspende o reanuda).

Como resultado, un sistema operativo corriendo Linux ofrece el soporte necesario para crear un entorno totalmente adaptado a las necesidades del usuario. Todo esto independientemente del sistema informático del computador anfitrión. Por ejemplo, puede ocurrir que una aplicación requiera de una librería en específico para funcionar, pero que la máquina anfitrión no la posea y sea complicado instalarla por incompatibilidades con otros programas del sistema. En este caso, se podría crear un `namespace` para aislar el software del anfitrión del sistema huésped y de esta manera instalar dentro del espacio virtualizado todo el software que se necesite sin conflictos.

Claro está, los contenedores tienen ciertas limitaciones. Para empezar, éstos sólo están implementados a partir de la versión 3.8 del kernel de Linux. Esto significa que las versiones viejas

u otros kernels de sistemas operativos, por ejemplo Mac o Windows, no los soportan. Luego, los contenedores no ofrecen virtualización hardware como sucede con la virtualización completa. Mediante contenedores no se pueden virtualizar los componentes hardware de un sistema: controladores, procesador, periféricos, etc. En consecuencia, los contenedores están sujetos a restricciones y su uso no puede ser viable en todo tipo de situaciones. Por ejemplo, resulta inútil instalar dentro de un contenedor una aplicación que necesite aceleradores, si el sistema donde se ejecutará no dispone de GPUs. O bien, uno no puede utilizar un contenedor con aplicaciones compiladas para una arquitectura de procesador en otra arquitectura incompatible.

1.3. Formulación del Problema

Gracias a la aparición de los contenedores, el sector HPC está experimentando una ola de optimismo, ya que éstos pueden servir como medio para portar aplicaciones entre centros HPC. Un ejemplo de este optimismo son las actividades conjuntas de investigación respaldadas por el proyecto de la Comisión Europea HPC-Europa3. Aún así, la virtualización a nivel de sistema operativo está pendiente de evaluación y la comunidad HPC mantiene lazos con las viejas costumbres y desconfía. Retos como: *portabilidad* "Como encapsular una aplicación y sus dependencias?"; *rendimiento* "Cómo podemos contener una aplicación y a la vez sacar un rendimiento nativo?"; *seguridad* "Es seguro el uso de contenedores en supercomputadores?"; *reproducibilidad* "Como podemos replicar los resultados de una simulación?". Éstas son unas pocas cuestiones técnicas que las tecnologías de contenedores tienen pendiente responder.

En este panorama, la tecnología de contenedores sugiere ser una solución factible para el sector HPC, pero no definitiva, pues carece de estudios que la avale. Esta razón ha motivado nuestro trabajo de evaluar el rendimiento y portabilidad de distintas implementaciones de contenedores en varias arquitecturas. Como ya se han hecho evaluaciones similares (tal como se expone en el Capítulo 2), mostrando resultados prometedores con *benchmarks*, nosotros vamos a ir un paso más allá y realizar este estudio no sólo con *benchmarks* sintéticos, sino con un programa en producción de simulación biológica llamado Alya, el cual se basa en cientos de miles de líneas de código. En definitiva, nuestros problemas formulados son los siguientes:

1. ¿Qué tecnología o implementación de contenedores se adapta mejor a los requisitos y necesidades del *High-Performance Computing*?

2. ¿Cuál es el rendimiento que se obtiene al ejecutar aplicaciones dentro de contenedores?

3. ¿Cuán portables son los contenedores entre diferentes sistemas?

4. ¿Qué relación existe entre lo portable que es un contenedor y el máximo rendimiento que ofrece?

5. ¿Resulta viable almacenar y ejecutar desde de un contenedor una aplicación científica real?

6. ¿Qué métodos puede seguir un usuario para obtener los mejores resultados posibles (en términos de rendimiento, reproducibilidad y portabilidad) al usar contenedores?

Con este trabajo, se espera contribuir a la literatura actual de los contenedores y motivar

su uso en el sector HPC.

1.4. Agentes Implicados

En esta sección se describen tanto las personas involucradas en el trabajo como los beneficiarios de éste.

Investigador

El investigador es la persona encargada ejecutar los experimentos, procesar los resultados y reportarlos. Además, también juega un papel en la toma de decisiones, dónde puede hacer sugerencias y observaciones al equipo directivo a la hora de diseñar los experimentos o revisar los resultados. Por último, pero no menos importante, también será responsable de las tareas de gestión del proyecto.

Dirección

La dirección está formada por el director y codirectora del proyecto. Estos dos agentes guiarán y supervisarán al investigador asegurando el correcto desempeño del proyecto.

Soporte

Como el trabajo involucra usar una amplia cantidad de máquinas HPC, es imprescindible contar con el apoyo del equipo de soporte de cada clúster (Lenox, MareNostrum4, CTE-POWER y ThunderX). Éstos informarán al investigador sobre el estado de las máquinas y lo ayudarán con las instalaciones del software necesario.

Es necesario mencionar la propia empresa BSC, la cual ofrecerá los recursos necesarios para llevar a cabo del trabajo a través de sus instalaciones, material, contratos y profesionales.

Beneficiarios

El objeto de estudio de este trabajo puede interesar a toda la comunidad HPC. Administradores de sistemas, mánagers de instalaciones HPC, ingenieros o expertos HPC y hasta científicos se pueden beneficiar del uso de los contenedores como herramienta de trabajo. Aparte, este trabajo también servirá para cumplir con las tareas del proyecto HPC-Europa3.

Capítulo 2

Estado del arte

Antes de desarrollar este TFG se ha repasado profundamente la literatura actual respecto al uso de contenedores en entornos HPC. En esta sección se resume el trabajo más relevante llevado a cabo juntamente con las carencias que nosotros hemos identificado. Se notará como después de cada análisis de la literatura se exponen cuáles son nuestras contribuciones.

Para empezar, en Bachiega et. al. [5] (un sondeo que revisa las recientes investigaciones sobre contenedores) concluyen que en la literatura actual hay escasez de estudios enfocados a entornos HPC y evaluación mediante aplicaciones reales en producción. Dada esta observación, nuestro trabajo trata de corregir estas dos carencias: primero, realizando los experimentos en centros HPC con clústers representativos; y segundo, ejecutando con contenedores Alya, una aplicación científica en producción.

Siendo más concretos, en la introducción hemos destacado que la portabilidad y el rendimiento es una de las principales razones para usar la virtualización a nivel de sistema operativo. Justamente Kóvacs [6] compara el rendimiento de los software de virtualización Docker [7], LXC [1], Singularity [8] y KVM [9] con ejecuciones nativas. Sin embargo, esta comparativa es realizada en sistemas poco representativos (16 cores) y usando *benchmarks* muy básicos (Sysbench e Iperf). Al final, se demuestra que la ejecución de programas dentro de contenedores puede igualar los rendimientos nativos. Nuestro trabajo trata de complementar la contribución de Kóvacs efectuando experimentos en grandes supercomputadores (hasta 12 mil cores) con una aplicación de simulación biológica de cientos de miles de líneas de código.

Otras evaluaciones de las implementaciones de contenedores las podemos encontrar en Hale et. al. [10] y Younge et. al. [11] donde prueban Shifter [12] y Singularity en sistemas HPC. Ambos trabajos prueban y comparan ejecuciones nativas con otras usando los contenedores en supercomputadores (el supercomputador Edison del centro NERSC y el Volta de Sandia National Laboratories). Sus experimentos se basan en la ejecución de *benchmarks* sintéticos y herramientas de computación, tales como HPCG, HPGMG-FE, IMB, y la caja de herramientas FeniCS. Investigaciones similares se pueden encontrar en [13], [14] y [15], cada una de ellas midiendo el rendimiento de los contenedores en HPC. Inspirándonos en estas publicaciones, nuestro trabajo ha extendido sus hallazgos *i)* evaluando la implementación de contenedores Docker *ii)* ampliando sus estudios de portabilidad usando diferentes clústers con distinta arquitectura de

[1] Más información sobre LXC: https://linuxcontainers.org/

procesador *iii)* integrando con contenedores una compleja aplicación científica en producción.

A parte de las evaluaciones, encontramos que en el dominio de la biología computacional ya se han probado los contenedores con programas científicos reales. O'Connor et. al. [16] presenta un caso de uso real enfocado al análisis de ADN y ensamblaje de genomas. En el artículo, los autores presentan Dockstore, un plataforma que estandariza las herramientas de Docker para un despliegue portable en la nube o entornos HPC. A raíz de ese artículo, en nuestro trabajo no sólo estudiamos la portabilidad mediante contenedores, sino también su rendimiento. Es más, aquí se discute el compromiso entre portabilidad/rendimiento durante el proceso de creación de los contenedores

Finalmente, Belkin et. al. [17] presenta un estudio de contenedores mucho más completo con computadores Cray. Ellos se centran en desplegar para producción la implementación de contenedores Shifter en el supercomputador Blue Waters, recolectar estadísticas de su uso y compararlas con ejecuciones nativas. Considerando su trabajo, aquí se va a investigar la viabilidad de diversas implementaciones de contenedores (no únicamente Shifter) en entornos HPC y valorar su impacto en términos de portabilidad y rendimiento.

Por lo tanto, el presente proyecto no sólo va a comparar distintas alternativas de contenedores a la vez (Docker, Singularity y Shifter) en entornos HPC reales y representativos, sino también evaluará qué portabilidad cabe esperar de los contenedores al intentar ejecutar una aplicación científica real como es Alya en diferentes arquitecturas. Por supuesto, todo esto se desarrolla mientras se compara el rendimiento ofrecido por los contenedores respecto las ejecuciones nativas (sin virtualización).

Capítulo 3

Descripción de las Tecnologías de Virtualización

En este trabajo se han evaluado 3 implementaciones de contenedores distintas: Docker, Singularity y Shifter. Por un lado, Docker porque es el estándar *de-facto* y tiene mucho peso en la comunidad. Por el otro lado, Singularity y Shifter porque ambas son implementaciones especialmente diseñadas para HPC, además de ser las más relevantes en la literatura. Cabe decir que existen muchas más implementaciones, como por ejemplo Charliecloud [18], LXC (Linux Containers) o rkt [1] (rocket). Aún así, consideramos que las 3 que hemos elegido son las más representativas.

En las siguientes secciones se describe cada implementación con sus características y funcionalidades más importantes. Al final de la sección se resume en la Tabla 3.1 las características de cada una.

3.1. Docker

Docker no es la primera implementación de contenedores, pero sí la más disruptiva pues reinventó el concepto de virtualización ligera y la manera de desarrollar software. Fue lanzado el año 2013 por Docker, Inc. y ofrece servicios tanto gratuitos y *open-source* como *enterprise*. Dado que Docker fue diseñado para el despliegue de microservicios en la nube, su implementación actual conlleva varios retos y riesgos de seguridad para los sistemas HPC. En la Figura 3.2 se muestra de manera muy simplificada la arquitectura de Docker.

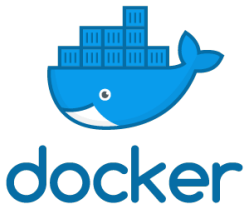


Figura 3.1: Logotipo de Docker.

El riesgo de seguridad primario consiste en que Docker usa un proceso *daemon* [2] con privilegios de administrador para desplegar sus contenedores. En consecuencia, los procesos dentro del contenedor

[1] Más información sobre rkt: https://coreos.com/rkt/

[2] Un tipo de proceso no interactivo que se ejecuta en segundo plano y que el usuario no tiene control directo sobre él.

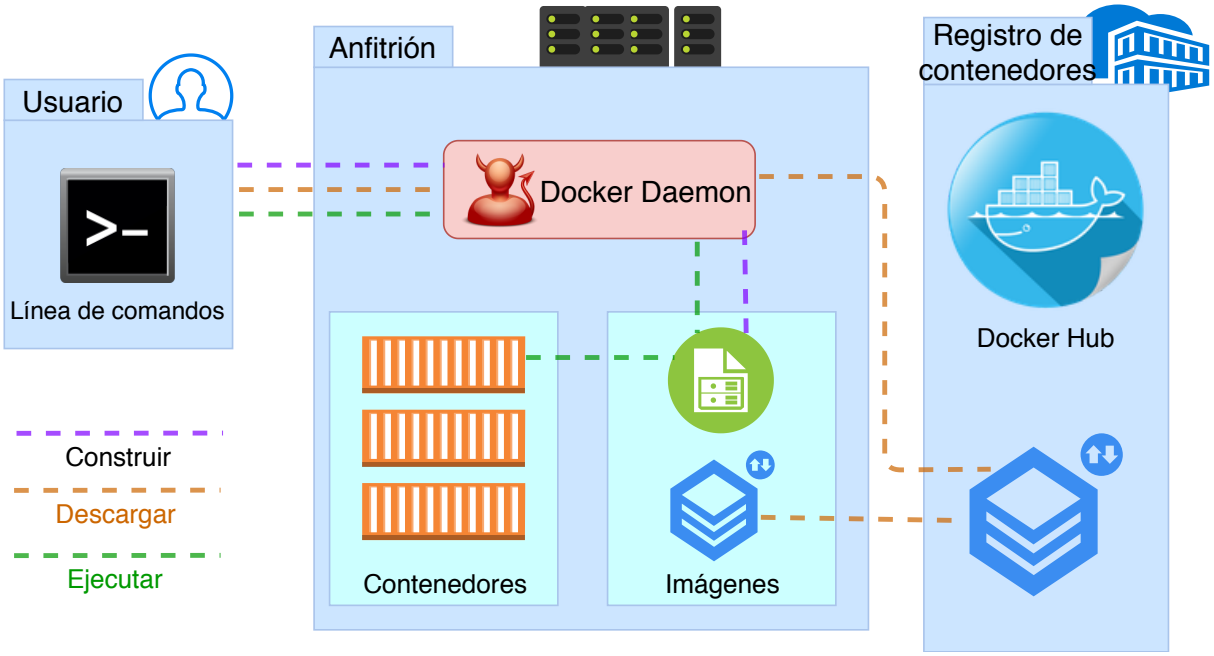


Figura 3.2: Modelo simplificado de la arquitectura de Docker.

tienen la potencial capacidad de escalar privilegios y amenazar el anfitrión desde el entorno virtual. Mientras que en el sector de las tecnologías de la información este hecho puede no suponer preocupaciones, pues cada empresa conoce qué tipo de código corre dentro de cada contenedor, en los centros HPC los usuarios son libres de ejecutar sus programas dentro de los contenedores. Como los administradores de sistemas no conocen *a priori* ni el código que van a ejecutar sus usuarios ni las intenciones de éstos, no se pueden permitir tal potencial problema de seguridad.

Por lo que respecta a su diseño, el objetivo de Docker es aislar completamente el sistema huésped del contenedor de los recursos del anfitrión. Esto se consigue gestionando tantos `namespace` y `cgroups` como sea posible, simplificando el control de los recursos disponibles para cada contenedor, pero complicando la integración de un contenedor con los componentes hardware específicos del anfitrión. Es decir, resulta muy fácil definir el porcentaje de CPU que cada contenedor tiene disponible, pero es complejo usar desde un contenedor la red del anfitrión sin antes virtualizarla.

Aún así, se han dedicado varios esfuerzos para adaptar Docker al sector HPC. La falta de integración de los contenedores de Docker con las librerías MPI (Message Passing Interface) [19] ha sido tratada en [20]. MPI es un modelo de programación paralelo basado en el envío de mensajes que sirve para arquitecturas de computadores con memoria compartida y distribuida. Se ha tomado una solución similar con el uso de las GPU's con NVIDIA-docker [3], una herramienta que facilita la creación de contenedores con el software necesario para un uso eficiente de las GPUs.

[3] Más información sobre NVIDIA-docker: https://github.com/NVIDIA/nvidia-docker

3.2. Singularity

Singularity aparece como un contenedor enfocado a suplir las necesidades del área HPC. Esta implementación fue declarada como proyecto *open-source* el año 2015 por los Laboratorios de Lawrence Berkeley y fue ganando mucha popularidad los años posteriores. Está programado para crear contenedores más integrados con los componentes del anfitrión a la vez que ofrece un entorno aislado donde ejecutar las aplicaciones. Además, Singularity es compatible con las imágenes de Docker.

Mientras Docker posee una compleja arquitectura para construir los contenedores (debido a su *daemon*), Singularity gestiona los permisos de SUID (Set User ID) para realizar las llamadas a sistema privilegiadas necesarias para la creación del contenedor. Una vez creado el contenedor, y antes de devolverle el control al usuario, Singularity abandona los privilegios de administrador y adopta los permisos del usuario que lo ha invocado. Asimismo, Singularity ofrece la posibilidad de desplegar contenedores sin necesidad de invocar llamadas a sistema privilegiadas a través del `namespace` de usuario (USR, aunque este método limita las funcionalidades del contenedor y su portabilidad).

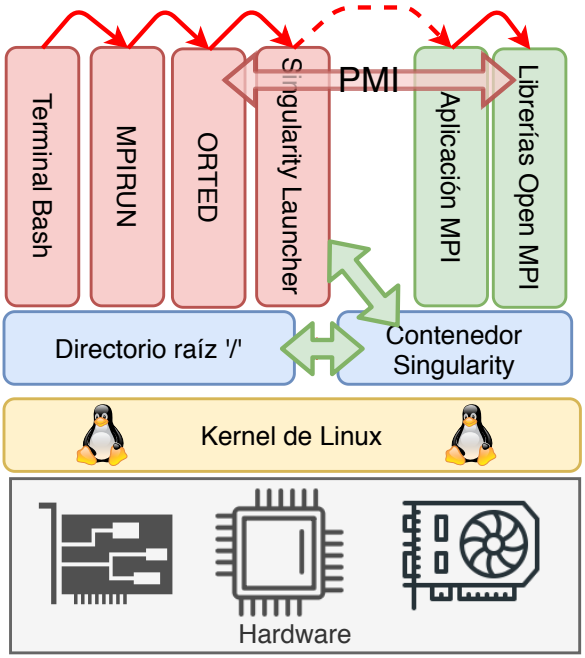


Figura 3.3: Interacción MPI Contenedor-Anfitrión.

Figura 3.4: Logotipo de Singularity

Dado que Singularity solamente gestiona por defecto los Mount y PID `namespace`, su uso resulta muy sencillo. Las llamadas MPI realizadas dentro del contenedor son comunicadas al *daemon* MPI del anfitrión (en esencia, el *daemon* invocado por la llamada de sistema `orted`) a través del PMI (Process Management Interface). En la Figura 3.3 se ilustra todo el proceso. Ésto hace que sea muy fácil ejecutar aplicaciones MPI, aunque hay una cierta restricción: las librerías MPI del contenedor y del anfitrión deben de ser compatibles. Igualmente, a partir de la versión 2.3 Singularity ofrece una funcionalidad para configurar automáticamente el uso de las GPUs del anfitrión en tiempo de ejecución.

Finalmente, Singularity es un software fácilmente instalable pues tiene pocas dependencias

y una configuración muy simple.

3.3. Shifter

Shifter se hizo público en diciembre de 2015 por NERSC (National Energy Research Scientific Computing Center) con el claro objetivo de ser la alternativa HPC de Docker. Shifter ataca los problemas encontrados en Docker considerando que los contenedores solamente deben tener permisos de usuario e integrar el entorno de ejecución del sistema anfitrión. Como resultado, Shifter trabaja con base en las imágenes de Docker, convirtiéndolas a un formato propio y aplicando parches de seguridad.

La arquitectura de Shifter está compuesta por 4 componentes bien diferenciados: la interfaz de la línea de comandos (CLI), el ImageGateway, el udiRoot y la integración con SLURM [4]. La CLI ofrece al usuario los comandos `shifter` y `shifterimg` para manejar los contenedores y sus imágenes respectivamente. El ImageGateway es un servicio que provee una base de datos para almacenar imágenes, una conexión con los repositorios de Docker y la capacidad de descargar y convertir las imágenes de Docker. Mientras que el ImageGateway se encarga de la gestión de las imágenes, el udiRoot ofrece unos scripts que crean los contenedores a partir de las imágenes y verifican su integridad antes de devolverle el control al usuario. Al igual que Singularity, Shifter adopta el método SUID para poder instanciar los contenedores. Finalmente, el componente de integración con SLURM ofrece la posibilidad de integrar las funcionalidades de Shifter con el gestor de colas, simplificando y optimizando su uso. En la Figura 3.6 se muestra el esquema de funcionamiento de los 4 componentes.

Figura 3.5: Logotipo de Shifter

Ejecutar aplicaciones con Shifter resulta igual de simple que con Singularity. Shifter utiliza la compatibilidad ABI (*Application Binary Interface*) de las distintas versiones y librerías MPI para comunicar las llamadas MPI del contenedor con las librerías específicas de su anfitrión. En caso que ésto no sea posible, los contenedores de Shifter directamente pasan las llamadas MPI al entorno de ejecución del anfitrión como con Singularity (siempre que las librerías MPI sean compatibles). En el caso de usar GPU's, se tiene la posibilidad de configurar Shifter para cargar en tiempo de ejecución todas las librerías y controladores necesarios dentro del contenedor.

[4] SLURM es un planificador de tareas muy usado en los centros HPC. Normalmente se usa para automatizar el cuándo y cómo se ejecutan las aplicaciones en los clústers.

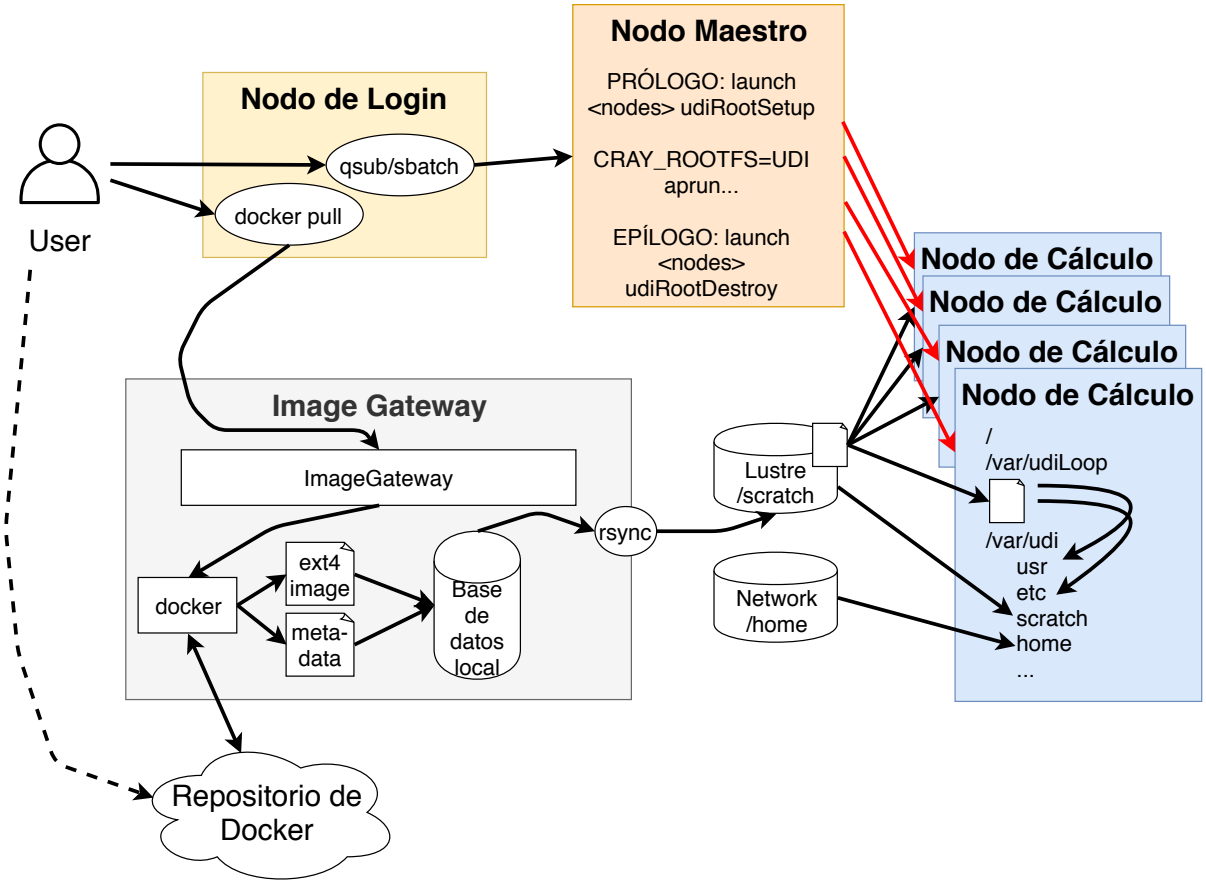


Figura 3.6: Flujo de trabajo de Shifter. Fuente: https://docs.nersc.gov/programming/shifter/overview/.

Característica	Docker	Singularity	Shifter
Diseñado para HPC	No	Sí	No
`namespace` utilizados	PID, NET, IPC, MNT, UTC	PID, MNT, USR	MNT
Uso de `cgroups`	Sí	No	No
Paradigma de seguridad	Root daemon	SUID/UserNS	SUID
Soporte MPI	No	Sí	Sí
Soporte GPU	No	Sí	Sí
Comptabile con gestores de colas	No	Sí	Sí
Instalación trivial	Sí	Sí	No

Tabla 3.1: Características principales de Docker, Singularity[a] y Shifter.

[a] Para versiones de Singularity 2.X.

Capítulo 4

Entorno de Pruebas

En este Capítulo se describe el hardware y software que se ha utilizado en el proyecto. Para empezar, se exponen las características de los clústers usados, que son: Lenox, MareNostrum4, CTE-POWER y ThunderX. A continuación, se explica el software usado para evaluar los contenedores y porqué. Este software consiste en: Alya, BT-MZ y HPCG.

4.1. Entorno Hardware

En la Tabla 4.1 se muestra un resumen de los clústers usados y sus características. Se ofrece una descripción más detallada en los siguientes apartados junto con la razón por la cual resultan útiles en este trabajo.

Clúster	#Cores/ Nodo	#Nodos	Red	Entorno				
				Linux Kernel	Open MPI	Docker	Singularity	Shifter
Lenox	28	4	1 Gb/s Ethernet	3.10.0	1.10.4	1.11.1	2.4.5	16.08.3
MareNostrum4	48	3.456	100 Gb/s Intel Omni-Path	4.4.12	1.10.4	N/A	2.4.2	N/A
CTE-POWER	40	52	100 Gb/s Mellanox Infiniband	4.11.0	3.0.0	N/A	2.5.1	N/A
ThunderX	96	4	40 Gb/s Ethernet	4.4.3	1.10.2	N/A	2.5.2	N/A

Tabla 4.1: Resumen del entorno de pruebas hardware.

4.1.1. Lenox

Lenox es un clúster con 4 nodos de cálculo propiedad de Lenovo donde poseemos privilegios de administrador. Cada nodo contiene una placa madre con 2 sockets, conteniendo cada socket un procesador Intel Xeon E5-2697v3 con 14 cores (28 cores por nodo). El clúster corre el kernel de Linux 3.10.0 con las librerías MPI Open MPI 1.10.4, Docker 1.11.1, Singularity 2.4.5 y Shifter 16.08.3. Los 4 nodos están conectados mediante una red 1Gb Ethernet.

Lenox nos resulta la plataforma ideal donde realizar la comparativa entre Docker, Singu-

larity y Shifter. Como podemos ejecutar `sudo` [1], hemos sido capaces de instalar y configurar los 3 entornos de contenedores por nuestros propios medios permitiéndonos comparar mejor si se adaptan a los entornos HPC. Conste que, instalar Docker en un superordenador en producción es inviable por sus problemas de seguridad, mientras que Shifter depende de muchos programas de terceros y una compleja configuración, lo cuál hace que su instalación sea poco viable en sistemas HPC muy grandes y complejos.

4.1.2. MareNostrum4

MareNostrum4 es un supercomputador en producción gestionado por Barcelona Supercomputing Center en Barcelona, España. Posee una cantidad total de 3.456 nodos disponibles, cada uno basado en 2 sockets, poseyendo cada uno un Intel Xeon Platinum 8160 CPU de 24 cores, 48 cores por nodo (165.888 cores en total). La red de interconexión MPI disponible es la red de alto rendimiento Intel Omni-Path 100 Gbit/s. Aparte, tiene una red Ethernet de 10 Gbit/s. MareNostrum4 ejecuta el kernel de Linux 4.4.12, con las librerías Open MPI 1.10.4, SLURM 17.11.7 como gestor de colas y posee una instalación funcional de Singularity 2.4.2.

Se usa este superordenador para realizar el test de escalabilidad con contenedores. Además, parte del estudio de portabilidad también se realiza aquí.

4.1.3. CTE-POWER

CTE-POWER es un clúster HPC también gestionado por el BSC. Este clúster está basado en procesadores IBM Power9 8335-GTG de 20 cores, donde cada nodo de cálculo contiene dos CPUs de las previamente mencionadas, proveyendo 40 cores por nodo. En total hay 52 nodos disponibles. CTE-POWER posee una red de alto rendimiento Mellanox Infiniband EDR 100 Gbit/s y otra red Ethernet de 10 Gbit/s. Su ecosistema software está formado por el kernel de Linux 4.11.0, Open MPI 3.0.0, SLURM 17.11.5 y Singularity 2.5.1.

Este clúster con procesadores de arquitectura IBM Power9 se usa en nuestro estudio de portabilidad.

4.1.4. ThunderX

El miniclúster ThunderX pertenece al proyecto Europeo Mont-Blanc [21] que desde el 2011 está introduciendo la arquitectura Arm en el sector HPC. El clúster posee 4 nodos de cálculo, cada uno con 96 cores Armv8-a organizados en dos sockets CN8890, con 48 cores cada uno, diseñados y manufacturados por Cavium/Marvell. Los nodos están conectados vía una red Ethernet de 40 Gbit/s. El clúster corre el kernel de Linux 4.4.3, con las librerías Open MPI 1.10.2, SLURM 17.11.3 y Singularity 2.5.2.

Como con CTE-POWER, ThunderX resulta muy útil para realizar el estudio de portabilidad con contenedores por tener una arquitectura de procesador diferente a la Intel x86.

[1] Un comando de Linux que permite ejecutar otro comando con privilegios de administrador.

4.2. Aplicaciones

En esta sección introducimos las aplicaciones que hemos usado para evaluar el rendimiento de los contenedores: Alya, BT-MZ y HPCG.

4.2.1. Alya

El código usado como ejemplo representativo de una aplicación científica real y en producción es Alya. Alya es una aplicación en Fortran desarrollada dentro del BSC para simulaciones numéricas de múltiples físicas. Este programa está formado por más de 500.000 líneas de código y es paralelizado mediante los modelos de programación MPI y OpenMP [22]. Por un lado, el modelo de programación MPI gestiona el paralelismo entre distintos nodos, por el otro, el modelo de programación OpenMP se encarga de paralelizar la carga de trabajo dentro de cada nodo. Como código paralelo específicamente diseñado para ejecutarse en ordenadores de alto rendimiento [23], [24], [25], forma parte de la UEABS [2] (Unified European Applications Benchmark Suite), una selección de 12 aplicaciones escalables, portables y relevantes para la comunidad científica. Alya usa el método de elementos finitos para la discretización espacial y el método de finitas diferencias para la discretización temporal.

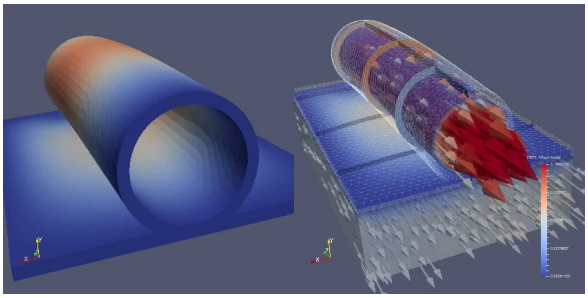
En este trabajo, se usara Alya para simular una interacción fluido-estructural (FSI, *Fluid-Structure Interaction*) de un sistema biológico. Este caso simula el flujo de la sangre a través de una arteria en contacto con otro tejido. A saber, en la Figura 4.1a se muestra el resultado generado por la simulación en un momento concreto del tiempo, donde se distinguen claramente las 2 físicas simuladas. En la parte derecha de la Figura 4.1a se simula del flujo de la sangre (el fluido mediante Dinámica de Fluidos Computacional). En esta parte de la ejecución se pretende calcular las físicas de la sangre como velocidad, presión, etc. Y en la parte izquierda de la imagen se simula la arteria (la parte sólida o estructural) representada con una malla. Aquí, se quiere ver como reacciona la arteria a consecuencia del flujo de la sangre, es decir, su deformación a lo largo del tiempo. Esta simulación estresa la CPU, red y memoria.

La Figura 4.1b muestra el dominio del problema. La estructura del sólido (el dominio con rayas de la Figura 4.1b) es un cilindro reposando sobre una superficie sólida. La parte del fluido (el dominio en gris de la Figura 4.1b) está compuesta por la parte interior del cilindro y el paralelepípedo en contacto con la superficie sólida. La dirección del flujo está representada con flechas.

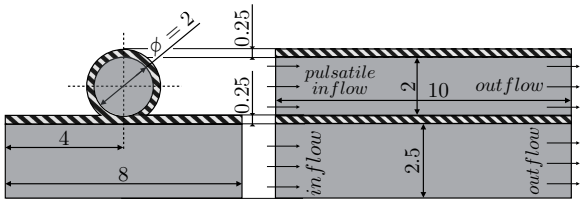
Para resolver este problema fluido-estructural se utiliza una aproximación multi-código. Los 2 dominios de la simulación (el fluido y sólido) quedan divididos en 2 subproblemas con una estrategia de descomposición Dirichlet-Neumann [26]. La estrategia para modelar cada subproblema está extensivamente descrita en [27], y el mecanismo computacional sólido (CSM) está basado en el modelo de [28]. La dinámica de fluidos computacional (CFD) dentro de las cavidades y vasos está modelada con el flujo de ecuaciones incomprimibles de Navier-Stokes. Estas ecuaciones están preparadas para resolver físicas de fluidos Newtonianos encima de una malla deformable utilizando el esquema ALE (*Arbitrary Lagrangian-Eulerian*).

Ambos modelos CSM y CFD+ALE son tratados como cajas negras, las cuales se comuni-

[2] Detalles sobre UEABS: https://repository.prace-ri.eu/git/UEABS/ueabs/



(a) Representación por computador de la simulación en un punto temporal. En la parte derecha se dibujan las físicas del fluido y en la parte izquierda la del sólido.



(b) Geometría del problema resuelto. La parte izquierda muestra el eje corto y la derecha una sección del eje largo.

Figura 4.1: Representaciones del caso de uso de Alya.

can sólo a través de las variables de interfaz de la descomposición Dirichlet-Neumann. Aunque ambas físicas son masivamente paralelas, son resueltas de manera secuencial por cuestiones de estabilidad. La física del sólido, la arteria, es resuelta con una instancia de Alya, la cual se comunica vía MPI con otra instancia encargada del fluido, la sangre. Ambos sistemas físicos requieren de una alta resolución espacial para captar eficazmente los detalles físicos, lo cual hace imprescindible el uso de equipo HPC. Cabe destacar que cada iteración del fluido y sólido está compuesto por 2 fases bien diferenciadas: la fase del ensamblaje, la cual no realiza comunicaciones MPI con los demás procesos y es mayormente cálculo; y la fase del solver, donde hay un elevado número de llamadas colectivas MPI. Gracias a estas 2 fases somos capaces de diferenciar las partes donde nuestras ejecuciones realizan cálculo y comunicación.

En la Figura 4.2 se ilustra el funcionamiento del algoritmo descrito. Cada fracción de tiempo (iteración) está formada por la ejecución del código MPI del fluido y sólido. A su vez, cada ejecución de las físicas está formada por la fase del ensamblaje y del solver. Todas las iteraciones tienen la misma cantidad de cálculo asignado para obtener unas medidas de escalabilidad precisas.

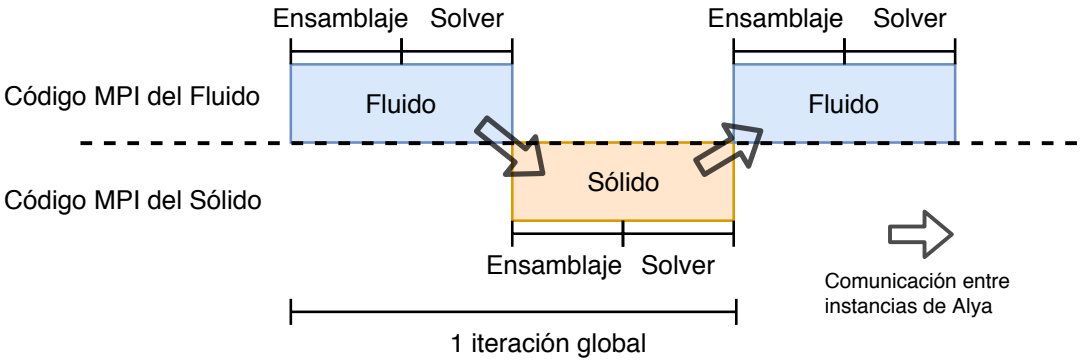


Figura 4.2: Esquema del algoritmo de Alya.

4.2.2. BT-MZ

BT-MZ 3.3.1 [29] pertenece al conjunto de *benchmarks* paralelos desarrollados por la NASA (NAS Parallel Benchmarks, NPB [3]). Estos están basados en algoritmos de Dinámica de Fluidos Computacional (CFD) enfocados a evaluar el rendimiento de los supercomputadores con mucho paralelismo. De todo el conjunto NPB, hemos elegido BT-MZ (Block Tri-diagonal solver Multi-Zone) porque soporta los modelos de programación MPI y OpenMP, haciendo que sea un *benchmark* capaz de explotar el paralelismo intra e inter-nodo y porque sus distintos hilos de ejecución no tienen una distribución de trabajo equitativa. La CPU, la red y el ancho de banda junto con la latencia de la memoria es lo que más afecta al rendimiento de BT-MZ.

Este *benchmark* resuelve una malla tri-dimensional de ecuaciones Navier-Stokes discretizadas. Se divide la malla en varias zonas 3D (con ejes x, y y z) y se reparten entre todos los procesos disponibles. Particularmente, en BT-MZ la malla está fragmentada de tal manera que las zonas poseen una cantidad de trabajo heterogéneo y crecen sólo en un eje. Como consecuencia, su ejecución presenta desbalanceo de trabajo entre procesos.

Los bucles están paralelizados con `PARALLEL DO` [4] y los datos son intercambiados mediante llamadas MPI no bloqueantes [5] mientras se usan barreras [6] para sincronizar todos los procesos. Cada iteración del BT-MZ está compuesta por 6 funciones:

1. Intercambio de datos con los procesos MPI vecinos.
2. Cálculo de la parte derecha de la matriz.
3. Resolver las ecuaciones de la dirección x.
4. Resolver las ecuaciones de la dirección y.
5. Resolver las ecuaciones de la dirección z.
6. Acumular los resultados.

```
1  do step 1, niter
2      call exch_qbc
3      do zone 1, num_zones
4          call compute_rhs
5          call x_solve
6          call y_zolve
7          call z_solve
8          call add
9      end do
10 end do
```

Código 4.1: Esqueleto del algoritmo de BT-MZ.

Los *benchmarks* del conjunto NPB ofrecen diversas clases de problemas. Cada clase de problema corresponde a un tamaño diferente de la malla que se resuelve. En la Tabla 4.2 mostramos

[3] Página web de los NPB: https://www.nas.nasa.gov/publications/npb.html
[4] Una directiva de OpenMP que automáticamente paraleliza los bucles de un código.
[5] Funciones que no bloquean el flujo del programa para sincronizar sus procesos.
[6] Funciones bloqueantes explícitas que ofrecen los modelos de programación paralelos para sincronizar los distintos hilos de ejecución de un programa.

el tamaño de la malla para cada clase de problema junto con la cantidad de zonas que genera.

Clase	Cantidad de zonas	Tamaños de la malla	Memoria requerida
S	2x2	24x24x6	1 MB
W	4x4	64x64x8	6 MB
A	4x4	128x128x16	50 MB
B	8x8	304x208x17	200 MB
C	16x16	480x320x28	0.8 GB
D	32x32	1632x1216x34	12.8 GB
E	64x64	4224x3456x92	250 GB
F	128x128	12032x8960x250	5.0 TB

Tabla 4.2: Tamaños de problemas del NPB BT-MZ.

Mediante este *benchmark* se van a aportar más datos sobre el rendimiento de las aplicaciones con contenedores.

4.2.3. HPCG

El *benchmark High-Performance Coarse Gradient* (HPCG) [30] está diseñado para emular el comportamiento de las aplicaciones científicas dedicadas a resolver ecuaciones en derivadas parciales (EDP). La implementación de HPCG es muy parecida al High-Performance LINPACK [7], excepto que HPCG se centra en estresar todos los componentes del computador y no sólo el procesador. HPCG trata de cubrir los patrones de comunicación y cálculo más comunes de las aplicaciones reales, poniendo especial interés en el uso de llamadas colectivas (por ejemplo, `MPI_Allreduce`) y el diseño de la memoria. De hecho, el máximo rendimiento de HPCG está limitado por la memoria.

El *benchmark* resuelve un gradiente conjugado precondicionado simétrico Gauss-Seidel con matrices dispersas. En la fase de inicialización, HPCG genera un modelo de ecuaciones 3D discretizado sintético. Este modelo luego es descompuesto con un gradiente conjugado para generar una matriz dispersa. Finalmente, HPCG itera sobre la matriz final tal como aparece en la Figura 4.3.

$p_0 \leftarrow x_0$, $r_0 \leftarrow b - Ap_0$
for $i = 1, 2,$ to max_iterations **do**
  $z_i \leftarrow M^{-1} r_{i-1}$
  if $i = 1$ **then**
    $p_i \leftarrow z_i$
    $\alpha_i \leftarrow$ dot_prod(r_{i-1}, z_i)
  else
    $\alpha_i \leftarrow$ dot_prod(r_{i-1}, z_i)
    $\beta_i \leftarrow \alpha_i / \alpha_{i-1}$
    $p_i \leftarrow \beta_i p_{i-1} + z_i$
  end if
  $\alpha_i \leftarrow$ dot_prod(r_{i-1}, z_i)/dot_prod(p_i, Ap_i)
  $x_{i+1} \leftarrow x_i + \alpha_i p_i$
  $r_i \leftarrow r_{i-1} - \alpha_i Ap_i$
  if $\|r_i\|_2 <$ tolerance **then**
    STOP
  end if
end for

Figura 4.3: Algoritmo del gradiente conjugado precondicionado de HPCG. Extraído de [30].

Como con BT-MZ, HPCG resulta útil para evaluar más a fondo el rendimiento de las tecnologías de contenedores.

[7] La aplicación usada para determinar el Top500 de los computadores más potentes del mundo.

Capítulo 5

Metodología y Rigor

En este capítulo se describe la metodología que hemos seguido para realizar el trabajo. Primeramente, describimos qué método sigue el TFG para asegurar su correcta ejecución. A continuación nos centramos en la metodología de los experimentos, contando qué métricas hemos extraído de las ejecuciones de Alya, BT-MZ y HPCG y mostrando qué proceso hemos seguido para construir las imágenes de los 3 contenedores evaluados.

5.1. Método de trabajo

Cuando se empezó el proyecto, el trabajo relacionado con contenedores en HPC era muy limitado. A raíz de este desconocimiento, elegir una metodología rígida y poco flexible resultaba arriesgado, pues fácilmente podían aparecer imprevistos o incompatibilidades porque existe poca información sobre el objeto de estudio. Por ello, se ha elegido el método de trabajo ágil Scrum [1], el cual se basa en definir objetivos simples en iteraciones temporales (en este caso cada 2 semanas). De esto modo, a lo largo del proyecto se han ido redefiniendo y adaptando los objetivos en base al conocimiento adquirido progresivamente.

5.1.1. Seguimiento

Para el seguimiento del proyecto se ha usado la herramienta gratuita y en línea Trello[2] además de planificar reuniones bisemanales con dirección. Trello es un gestor de tareas que permite crear, compartir y actualizar tareas. Entre reuniones bisemanales de cada iteración el investigador es el encargado de reportar vía Trello el estado de las tareas. Durante las reuniones, el investigador y dirección del proyecto revisan la página del Trello y evalúan el estado del trabajo.

[1] Más información acerca Scrum: https://proyectosagiles.org/que-es-scrum/
[2] Más información acerca Trello: https://trello.com/about

5.1.2. Validación

Para validar el trabajo se procura que los experimentos sean reproducibles. Para ello, no se usa software propietario (a excepción de Alya) asegurando así que cualquier persona podrá realizar los mismos experimentos en un entorno similar. Igualmente, en las reuniones bisemanales se revisa la ejecución de los experimentos y sus resultados procurando identificar errores.

Todos los resultados numéricos presentados son extraídos de varias repeticiones y de un entorno lo más estable posible. Es decir, nos hemos preocupado que los números sean estadísticamente correctos, por eso hacemos la media de las métricas y enseñamos su desviación estándar como prueba. Claro está, que no podemos realizar infinitas repeticiones de los experimentos, por lo que hemos elegido el valor que más se adecúa teniendo en cuenta nuestra planificación y la disponibilidad de los clústers HPC.

5.2. Experimentos

En esta sección se describen los experimentos realizados. En orden, estos experimentos consisten en:

1. **Comparativa entre implementaciones de contenedores**: se evalúan Docker, Singularity y Shifter comparándolos entre ellos y con el anfitrión (el escenario ideal). Este experimento se realiza exclusivamente en Lenox, pues es el único entorno donde tenemos disponibles las 3 tecnologías. Se valorará la instalación y configuración de cada contenedor, el coste de desplegar cada uno y finalmente el rendimiento que se obtiene al ejecutar aplicaciones con ellos. Las aplicaciones que se van a ejecutar son: Alya, BT-MZ y HPCG.

2. **Estudio de portabilidad entre arquitecturas**: se evalúa la portabilidad de los contenedores a través de 3 sistemas HPC con arquitecturas distintas: MareNostrum4 con Intel Skylake, CTE-POWER con IBM Power9 y ThunderX con ARMv8-A. Se trata de ejecutar una aplicación científica (Alya) usando una misma imagen de contenedor y medir el rendimiento que se obtiene comparándolo con el rendimiento ideal (el que se obtiene al ejecutar la aplicación directamente sobre el anfitrión).

3. **Test de escalabilidad**: se pretende poner a prueba la virtualización con contenedores ejecutando Alya con una gran cantidad de recursos en un supercomputador. Se calculará la ganancia en tiempo de ejecución obtenida y se valorará si las aplicaciones científicas escalables pierden rendimiento debido al uso de los contenedores.

En la Tabla 5.1 se resumen los experimentos realizados junto con los recursos utilizados y el objetivo.

Experimento	Entorno de pruebas hardware	Aplicaciones ejecutadas	Métricas medidas	Objetivo
Comparativa entre contenedores	Lenox	Alya (CFD), BT-MZ, HPCG	Tiempo de ejecución [s], FLOP/s	Comparar Docker, Singularity y Shifter. Valorar qué implementación se adapta mejor a entornos HPC.
Estudio de portabilidad	MareNostrum4, CTE-POWER, ThunderX	Alya (CFD)	Tiempo de ejecución [s]	Evaluar la relación entre portabilidad y rendimiento de los contenedores.
Test de escalabilidad	MareNostrum4	Alya (FSI)	Tiempo de ejecución [s]	Evaluar el máximo rendimiento obtenido con los contenedores.

Tabla 5.1: Tabla resumen de experimentos.

5.3. Métricas

Porque la simulación fluido-estructural (FSI) de Alya es muy pesada en términos computacionales, se ha decidido partir la simulación en 2 casos distintos.

- **CFD**: Simulación sólo de la física del fluido (la sangre). Requiere 2.58 veces menos tiempo de ejecución que el caso FSI completo. Sólo se ejecuta la instancia de Alya del fluido.
- **FSI**: Simulación fluido-estructural completa (sangre y arteria). Es el caso pesado dónde se ejecutan 2 instancias de Alya, una del fluido y la otra del sólido.

En nuestros experimentos de comparativa de contenedores y estudio de portabilidad se ejecutará el caso CFD de Alya. Así, minimizamos el uso de las máquinas y ahorramos en consumo energético. El caso FSI se ejecutará en el test de escalabilidad, pues es donde realmente nos interesa evaluar el rendimiento de los contenedores con un caso de uso real.

Ejecuciones típicas de la simulación de Alya suelen abarcar miles de fracciones de tiempo. En nuestro caso, descomponemos la duración de cada fracción de tiempo en $t_i = t_s + t_m$ midiendo las 2 fases de cómputo principales: t_s, la duración de la fase del solver, la cual posee una alta cantidad de colectivas MPI, y t_m, la duración del ensamblaje de la matriz, la que no posee comunicaciones MPI ni sincronizaciones. Puesto que los pasos de tiempo son algorítmicamente homogéneos, podemos expresar la duración media de las pasos como:

$$\overline{t_i} = \overline{t_s} + \overline{t_m}$$

Donde:

- **Duración de la fracción de tiempo**: $\overline{t_i} = \dfrac{\sum_{i=1}^{n} t_i}{n}$
- **Tiempo del solver**: $\overline{t_s} = \dfrac{\sum_{i=1}^{n} t_s}{n}$
- **Tiempo del ensamblaje**: $\overline{t_m} = \dfrac{\sum_{i=1}^{n} t_m}{n}$

Hemos considerado que para las ejecuciones de Alya una $n = 20$ es un buen valor para obtener un tiempo de simulación y una precisión estadística aceptable. En resumen, para Alya vamos a medir la duración de las fracciones de tiempo con las duraciones totales de la fase del

solver y ensamblaje.

Para el caso de BT-MZ y HPCG, las métricas las hemos extraído del informe final que nos reporta cada uno. Estas métricas son tiempo de ejecución del algoritmo principal y sus respectivos FLOP/s (operaciones de coma flotante por segundo). Hemos ejecutado BT-MZ con la clase de problema C 50 veces, un valor acorde teniendo en cuenta su bajo tiempo de ejecución. HPCG lo hemos ejecutado con unas dimensiones locales del problema de 120x120x120 (9.9 GB de memoria para datos requeridos) 25 veces, otra vez considerando su tiempo de ejecución.

5.4. Creación de las Imágenes

Para crear las imágenes de Docker, Singularity y Shifter hemos partido de una imagen común de Docker. Esta imagen base es una distribución de sistema operativo (Ubuntu, Debian) descargada de repositorios oficiales de Docker Hub [3]. Encima de esta imagen hemos instalado las librerías MPI sin soporte a redes de alto rendimiento (por ejemplo, Mellanox Infiniband o Intel Omni-Path). Porque Shifter no implementa una manera de crear imágenes, sino que toma las de Docker y las convierte a un formato particular mediante el comando `shifterimg pull`, la imagen que usa Shifter es la de Docker después de ser convertida. A diferencia de Shifter, Singularity sí que implementa una manera de crear imágenes a través del comando `singularity build`. Por esta razón hemos decidido construir la imagen final con nuestra aplicación usando las herramientas de Singularity partiendo de la imagen de Docker con la instalación de las librerías MPI. Todo el proceso de creación de las imágenes queda ilustrado en la Figura 5.1.

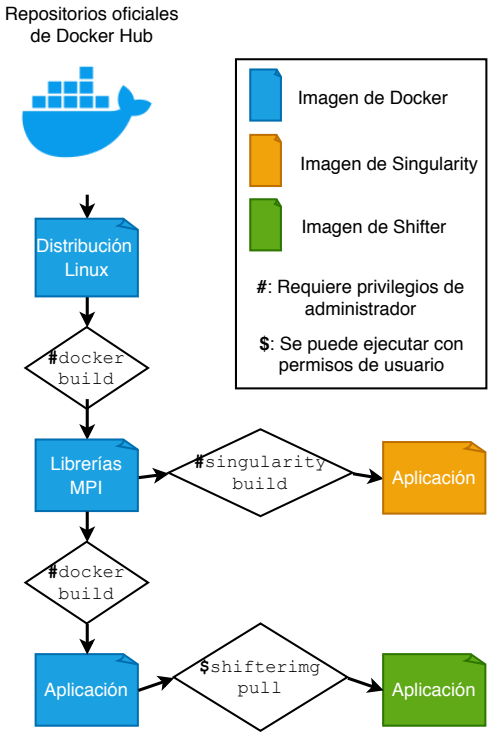


Figura 5.1: Proceso de creación de imágenes utilizado.

[3] Enlace a Docker Hub: https://hub.docker.com

Las imágenes de las aplicaciones resultantes de Docker, Singularity y Shifter contienen exactamente el mismo software, sólo varía el formato de la imagen. En la Tabla 5.2 se muestra el tamaño de las imágenes de Alya generadas, donde se nota que cada implementación de contenedores usa un formato distinto para sus imágenes. Ambos Singularity y Shifter utlizan el formato SquashFS [4] para almacenar sus imágenes, y como resultado tienen el mismo tamaño. Docker, por el contrario, utiliza un formato propio llamado *Docker Layers*. Este sistema de ficheros implementa copia-en-escritura, es decir, antes de modificar la imagen Docker copia su contenido y lo almacena en una estructura de sólo lectura. En consecuencia, la imagen de Docker es la más pesada ocupando 5.16 GB porque durante el proceso se ha modificado varias veces la imagen inicial.

	Tamaño [GB]	Formato
Docker	5.16	Docker Layers
Singularity	1.10	SquashFS
Shifter	1.10	SquashFS

Tabla 5.2: Tamaño y formato de la imagen de Alya con cada contenedor.

[4] SquashFS es un formato de ficheros comprimido de sólo lectura para Linux.

Capítulo 6

Comparativa entre Implementaciones de Contenedores

Actualmente existen varias implementaciones de contenedores, todas ellas enfocando distintos problemas y con diferentes maneras de usar los `namespace` y `cgroups`. Por ello, es difícil decidir arbitrariamente cuál es el mejor contenedor para HPC sin antes compararlos. Aún así, tampoco disponemos de recursos ilimitados para probarlos todos y debemos hacer una selección de los contenedores más representativos.

En este capítulo se va a comparar el proceso de instalación y configuración, usabilidad, coste de despliegue y liberación de recursos, y rendimiento de 3 implementaciones de contenedores: Docker, Singularity y Shifter. Todas las evaluaciones se realizan en Lenox, pues allí disponemos de privilegios de administrador. En los apartados a continuación mostramos todas la evaluaciones una por una.

6.1. Proceso de Instalación y Configuración

6.1.1. Docker

Debido a su gran popularidad, instalar Docker es una tarea muy sencilla. Docker ofrece múltiples repositorios y paquetes debian para automatizar la instalación, por lo que sólo ha sido necesario añadir el repositorio pertinente en nuestro gestor de paquetes (en este caso `yum`, pues Lenox posee la distribución Red Hat Linux) e instalar el paquete de Docker. Una vez instalado el paquete, es necesario levantar el *daemon* de Docker mediante `sudo service docker start`. A partir de aquí, ya se dispone de una instalación de Docker funcional, puesto que su configuración por defecto es suficiente.

6.1.2. Singularity

La instalación de Singularity, parecida a Docker, también resulta sencilla y directa. Al ser un software respaldado por la empresa Sylabs [1] y su comunidad de ususarios, ofrece un proceso de instalación compatible con la vasta mayoría de distribuciones Linux. La mejor manera de instalar Singularity ha sido compilando el código disponible en su repositorio de GitHub [2] ya que tiene pocas dependencias con otras librerías. Una vez compilado el código e instalados los binarios, su configuración por defecto ya sirve para ejecutar los contenedores de manera efectiva.

6.1.3. Shifter

En el caso de Shifter, la instalación ha resultado ser muy compleja y larga. El código de Shifer está disponible en su repositorio de GitHub [3] junto con varios ficheros de documentación. Esta documentación, sin embargo, está dispersa entre todo el directorio del repositorio, es escasa, poco clara y muchas veces desactualizada o a medio terminar. Además de la pobre documentación que hay sobre Shifter, sus múltiples dependencias con otro software complica toda la instalación y configuración. En la Tabla 6.1 se muestran todas las dependencias de Shifter.

```
rpm-build gcc glibc-devel munge libcurl-devel json-c json-c-devel pam-devel
munge-devel libtool autoconf automake gcc-c++ python-pip xfsprogs squashfstools
python-devel libcap-devel python-flask python-gunicorn python-pymongo
```

Tabla 6.1: Dependencias de Shifter.

La instalación de Shifter está compuesta por 2 partes: 1) la instalación del ImageGateway, la parte que se encarga del almacenamiento y gestión de las imágenes; 2) la instalación del entorno de ejecución de Shifter, la parte responsable de leer las imágenes, verificarlas y desplegarlas. El ImageGateway resulta especialmente complejo de instalar, pues requiere de 3 servicios externos para funcionar:

1. Una base de datos de contenedores basada en MongoDB [4].
2. El servicio de autentificación ofrecido por MUNGE [5].
3. Un servidor HTTP mediante Gunicorn [6].

Finalmente, la configuración de Shifter no es trivial. El fichero de configuración por defecto no sirve y se requiere adaptarlo para el sistema anfitrión. Todos los detalles sobre el proceso de instalación y configuración de Shifter se pueden encontrar en el apartado 3.4 del informe técnico [31], trabajo resultante del proyecto HPC-Europa3.

[1] Sylabs: https://www.sylabs.io
[2] Repositorio de Singularity: https://github.com/sylabs/singularity
[3] Repositorio de Shifter: https://github.com/NERSC/shifter
[4] MongoDB es un sistema de bases de datos NoSQL: https://www.mongodb.com/.
[5] MUNGE es un servicio de autentificación y validación de credenciales entre computadores: https://dun.github.io/munge/.
[6] Gunicorn ofrece un servidor HTTP para Unix implementado con Python: https://gunicorn.org/

6.1.4. Conclusiones

Después de realizar personalmente todo el proceso de instalación y configuración de los 3 contenedores podemos concluir lo siguiente.

- La instalación de Docker y Singularity resulta ser muy sencilla. Son contenedores lo suficientemente desarrollados para poder ejecutarse en prácticamente cualquier sistema. Además, tienen pocas dependencias con otro software, lo que facilita enormemente su mantenimiento.

- Dado que Docker utiliza un proceso *daemon* para funcionar, requiere un poco más de configuración que Singularity. Asimismo, como este *daemon* requiere permisos de administrador, es necesario prestar especial atención a qué usuarios pueden interactuar con él para evitar problemas de seguridad.

- Shifter ha devenido el contenedor más díficil de instalar. Su pobre documentación junto con la gran cantidad de dependencias hace improbable lograr una instalación funcional al primer intento. Por si fuera poco, Shifter padece de numerosos bugs y no parece estar completamente implementado, pues algunas funcionalidades básicas, como por ejemplo borrar imágenes descargadas de su base de datos, no existen. En [17] se relatan más problemas encontrados a la hora de utilizar Shifter.

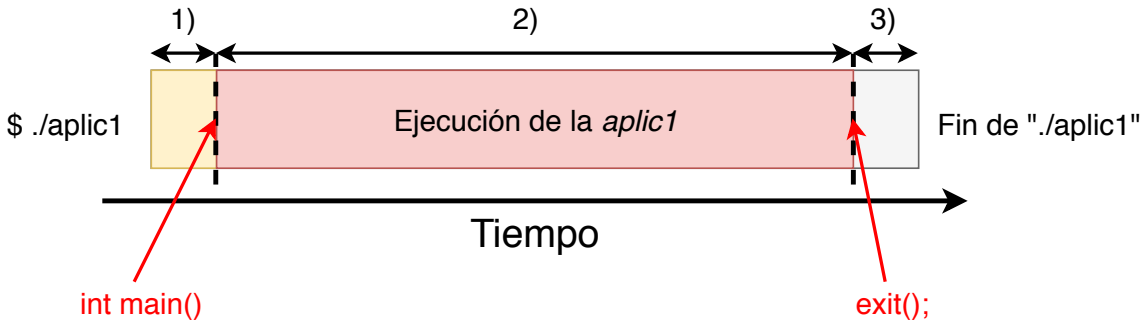
6.2. Coste de Despliegue

El coste de despliegue es el tiempo que se tarda en inicializar y liberar todos los recursos necesarios antes y después de ejecutar cualquier aplicación. En la Figura 6.1 se representa el coste de despliegue de una tarea, dónde se ve como la ejecución de una aplicación no es instantánea, sino que antes y después el entorno de ejecución o el kernel necesitan reservar los recursos y liberarlos al terminar (el sistema de colas por ejemplo).

Este coste de despliegue puede resultar insignificante cuando se ejecuta el programa directamente sobre el anfitrión (ejecución nativa), pero es un hecho que el uso de contenedores lo aumenta. Al usar contenedores, para ejecutar la aplicación se tiene que virtualizar el entorno (crear el contenedor y destruirlo al terminar), por lo que el tiempo de ejecución total aumenta. Justamente por esto, es necesario evaluar el coste de despliegue de los contenedores respecto a ejecuciones nativas. Si el coste resulta ser significativo, el uso de contenedores empeora la eficiencia de los sistemas HPC.

6.2.1. Metodología

Para evaluar el coste de despliegue con contenedores hemos diseñado el siguiente experimento. Vamos a ejecutar con Docker, Singularity y Shifter el código de los Códigos 6.1 y 6.2. El programa del Código 6.1 es totalmente secuencial, mientras que el del Código 6.2 está programado con MPI para ser ejecutado en paralelo con varios procesos. A estos programas los hemos llamado `sleep`, ya que solamente se duermen durante 5 segundos y finalizan. Por lo tanto, podemos considerar como coste de despliegue cualquier tiempo añadido a los 5 segundos que los



1): Tiempo que se tarda en ejecutar el *main()* de la aplicación desde que se ordena.

2): Tiempo que tarda la aplicación en ejecutarse.

3): Tiempo que se tarda en liberar los recursos utilizados desde que la aplicación termina.

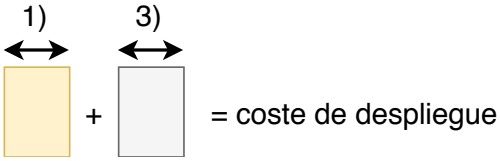


Figura 6.1: Representación del coste de despliegue.

programas se quedan dormidos. El código secuencial nos sirve para medir el coste de desplegar 1 contenedor, mientras que con el paralelo podemos ver como varía el coste si aumentamos el número de procesos.

```c
#include<unistd.h>
#include<stdlib.h>
int main(int argc, char *argv[]) {
    sleep(5);
    exit(EXIT_SUCCESS);
}
```

Código 6.1: Programa `Sleep` secuencial.

```c
#include "mpi.h"
#include <stdlib.h>
#include <unistd.h>
int main(int argc, int *argv[]) {
    MPI_Init(&argc, &argv);
    sleep(5);
    MPI_Finalize();
    exit(EXIT_SUCCESS);
}
```

Código 6.2: Programa `Sleep` MPI.

Hemos ejecutado 26 veces [7] estos 2 programas con cada implementación de contenedor y nativamente (sobre el anfitrión) para tener un punto de referencia. Hemos usado el comando `time` para medir el tiempo total que tarda en ejecutarse la tarea. Al tiempo reportado por `time` le restamos los 5 segundos de la función `sleep()` y declaramos el tiempo restante como coste de despliegue.

Lenox no dispone de ningún sistema de colas, por lo que hemos ejecutado los programas manualmente cuando ningún usuario estaba ocupando la máquina para evitar ruido de sistema. Como Singularity y Shifter ofrecen soporte MPI, esto es, son capaces de interactuar con el anfitrión para desplegar tantos procesos como sean necesarios, ejecutar los programas con ellos

[7] 26 repeticiones ofrecen una buena fiabilidad estadística y no consumen demasiado tiempo de CPU.

es muy simple. Con Docker, sin embargo, nos hemos visto obligados a desplegar un clúster virtual antes de ejecutar nuestros programas. En la Tabla 6.2 resumimos como hemos ejecutado nuestros programas con cada tecnología.

Tecnología	Sleep	MPI Sleep
Nativo	time ./sleep	time mpirun -np $PROCS ./mpi_sleep
Singularity	time singularity exec $IMAGEN ./sleep	time mpirun -np $PROCS singularity exec $IMAGE ./mpi_sleep
Shifter	time shifter –image=$IMAGEN ./sleep	time mpirun -np $PROCS shifter –image=$IMAGE ./mpi_sleep
Docker	(desplegamos un clúster virtual de Docker en Lenox)	(desplegamos un clúster virtual de Docker en Lenox)

Tabla 6.2: Forma simplificada de ejecutar los programas con cada tecnología en Lenox.

La diferente manera de desplegar las aplicaciones con Docker es debido al paradigma totalmente opuesto que adopta respecto Singularity o Shifter. El objetivo de Docker es aislar lo máximo posible el contenedor del anfitrión, por lo que el anfitrión no puede comunicarse con el contenedor para sincronizar sus procesos o compartir sus recursos. Aún así, lo que marca la diferencia es que Docker no ofrece soporte MPI, lo cual nos obliga a crear un clúster virtual de contenedores para poder ejecutar nuestros programas. Para esto, debemos crear un contenedor de Docker en cada nodo de Lenox, y luego crear una red virtual para interconectarlos puesto que no podemos usar la del anfitrión. Para crear esta red, hemos habilitado una red *multi-host* virtual [8] con Etcd. Etcd [9] es un software para almacenar datos tipo *clave-valor* en sistemas distribuidos, que en este caso resulta útil para almacenar los datos de red de los contenedores. Como resultado, los paquetes que se envían los contenedores de Docker a través de esta red quedan encapsulados en una cabecera VXLAN (la red virtual), la cual permite comunicar las capas 2 y 3 del modelo OSI (ver Figuras 6.2 y 6.3), y delega a Etcd la tarea de enrutar los paquetes a las direcciones de los contenedores correspondientes.

En resumen, para desplegar una aplicación con Docker hemos seguido los siguientes pasos. Estos pasos los hemos realizado para los 2 códigos porque en todos ellos hemos querido tener acceso a todos los recursos computacionales de Lenox.

1. Crear una red virtual con Docker.
2. Levantar un contenedor en cada nodo de Lenox (4 contenedores, uno por nodo).
3. Conectarnos a un contenedor y ejecutar la aplicación.
4. Una vez la ejecución ha terminado, eliminar todos los contenedores levantados.
5. Eliminar la red virtual.

[8] Todos los detalles sobre la creación de la red de Docker con Etcd: https://docker-k8s-lab.readthedocs.io/en/latest/docker/docker-etcd.html

[9] Información sobre Etcd: https://github.com/etcd-io/etcd

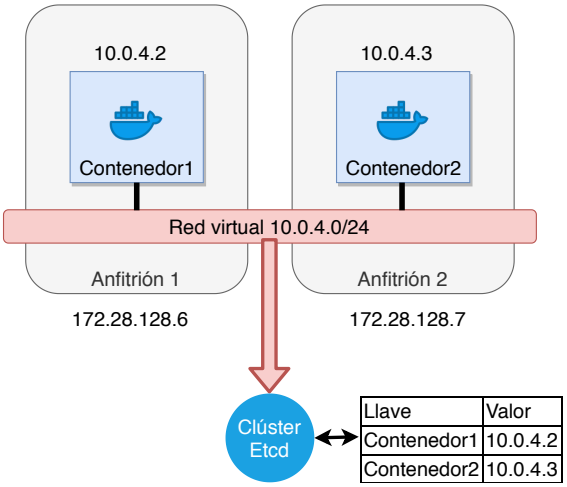


Figura 6.2: Esquema de la red virtual de Docker.

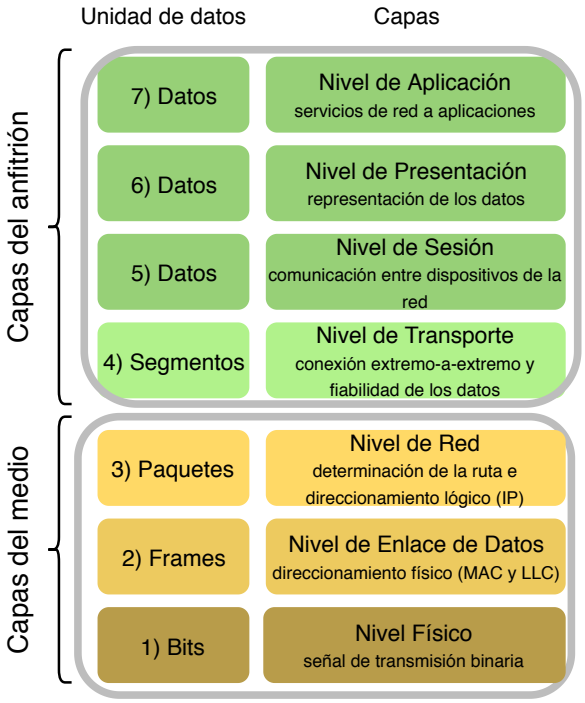


Figura 6.3: Modelo de Interconexión de Sistemas Abiertos (OSI). Origen: Wikipedia.

6.2.2. Resultados

La Tabla 6.3 contiene los resultados de estos experimentos con la desviación estándar (SD) de las 26 medidas. El coste de desplegar nativamente el programa secuencial (parte izquierda de la Tabla 6.3) es 0, lo cual tiene sentido puesto que el kernel toma el control directamente. En el caso de Singularity y Shifter, sin embargo, debemos crear los contenedores y destruirlos al acabar, lo que provoca que el coste llegue a 40 milisegundos con Singularity y 190 milisegundos con Shifter. Si ahora ejecutamos el programa paralelo MPI (parte derecha de la Tabla 6.3) escalando la cantidad de procesos desde 8 hasta 112 (todos los cores de los 4 nodos de Lenox están ejecutando el mismo proceso), vemos como el coste de despliegue aumenta proporcionalmente. El coste nativo ya no es 0, siendo esto lógico pues ahora nuestro programa debe interactuar con el entorno de ejecución de MPI. Los costes de Singularity parecen escalar idénticamente que las ejecuciones nativas, pero con un sobrecoste constante de alrededor de 0.2 segundos. El coste de Shifter también incrementa con la cantidad de procesos, aunque éste parece tener un crecimiento lineal: de 28 a 56 procesos el coste aumenta 0.3 segundos, pero de 56 a 112 aumenta 0.8 segundos. Con Docker los costes se disparan y no se pueden comparar con las otras tecnologías. Con el programa secuencial el coste llega a los 91 segundos, y con el paralelo se mantiene constante alrededor de los 104 segundos. La razón por la que Docker presenta costes tan altos es debido a su complejidad a la hora de desplegar aplicaciones, tal como está explicado en el apartado de la metodología 6.2.1.

Finalmente, puede parecer que el coste de despliegue con Docker en el caso MPI no incremente proporcionalmente con la cantidad de procesos. Hay que tener presente que con Docker el tiempo para crear sus 4 contenedores es siempre constante, a diferencia de Singularity y Shiter los cuáles crean tantos contenedores como procesos MPI especificados, y que la desviación

	Sleep		MPI Sleep									
			8		16		28		56		112	
	Tiempo [s]	**SD**	**Tiempo [s]**	**SD**	**Tiempo [s]**	**SD**	**Tiempo [s]**	**SD**	**Tiempo [s]**	**SD**	**Tiempo [s]**	**SD**
Nativo	0.00	0.00	0.98	0.02	0.97	0.02	1.01	0.02	1.06	0.03	1.20	0.03
Singularity	0.04	0.00	1.15	0.03	1.17	0.03	1.20	0.02	1.25	0.02	1.45	0.03
Shifter	0.19	0.01	1.42	0.04	1.51	0.04	1.60	0.05	1.90	0.03	2.70	0.09
Docker	91.16	3.04	104.94	2.68	103.58	0.56	103.83	0.57	103.67	0.46	103.94	0.65

Tabla 6.3: Costes de despliegue con cada tecnología.

estándar es considerable. Por lo tanto, creemos que sí hay un incremento del coste al escalar en procesos, pero este es insignificante comparado con el coste de crear la red, los contenedores y limpiar todo el entorno al terminar.

6.2.3. Conclusiones

Hemos podido comprobar que, efectivamente, el uso de contenedores afecta negativamente al coste de despliegue. Por un lado, los costes de Singularity y Shifter son lo suficientemente pequeños para permitirlos, aunque claramente la implementación de Shifter es peor que la de Singularity por 2 razones. La primera, es que Shifter tarda 150 milisegundos más que Singularity en desplegar un sólo contenedor. La segunda, es que Shifter muestra un crecimiento lineal en el coste de despliegue con programas MPI.

Por el otro lado, el coste de Docker es demasiado elevado al igual que su complejidad, más si se trabaja con aplicaciones MPI ejecutándose en varios nodos. Además, en este experimento sólo teníamos disponibles 4 nodos, pero en un superordenador con centenares o miles de nodos resulta impensable desplegar una aplicación con Docker usando este método. En un caso así, desplegar la infraestructura virtual de los contenedores y red no es nada trivial, tanto desde el punto de vista de tiempo como complejidad.

6.3. Rendimiento

Con el fin de comparar el rendimiento de los contenedores, entre ellos y el entorno nativo, presentamos a continuación los resultados obtenidos ejecutando Alya, BT-MZ y HPCG. Se omite la metodología, pues queda explicada en el Capítulo 5 qué métricas hemos medido para cada aplicación, y en el apartado 6.2.1 cómo hemos ejecutado las aplicaciones con cada tecnología (ver Tabla 6.2).

Todas las ejecuciones en esta subsección utilizan los 112 cores de Lenox y prueban distintas distribuciones MPI+OpenMP para ver como influyen en el rendimiento.

6.3.1. Resultados de Alya

En la Figura 6.4 aparece el tiempo medio transcurrido de cada fracción de tiempo con cada versión (nativo, Docker, Singularity y Shifter) del caso FSI usando los 4 nodos de Lenox (112 cores). En el eje de las abscisas se dibujan distintas distribuciones de procesos MPI y hilos por

proceso. Por ejemplo, 8x14 significa que Alya se ha ejecutado con 8 procesos MPI y que cada proceso dispone de 14 hilos (112 hilos de ejecución en total, 1 hilo en cada core). En el eje de las ordenadas aparecen representadas usando barras la medias aritméticas del tiempo transcurrido y de su desviación estándar mediante segmentos.

Podemos observar como las versiones nativa, Singularity y Shifter obtienen tiempos idénticos con variaciones muy parecidas. Por el contrario, Docker mantiene afín sus tiempos con la versión nativa hasta 28x4, a partir de donde pierde rendimiento hasta la distribución 112x1 en que indudablemente sufre algún tipo de problema. Las distribuciones 28x4 y 56x2 muestran una intensa variabilidad, aún así, como esta desviación también ocurre en la versión nativa parece ser algo inherente de Alya. Esta variabilidad puede ser efecto de una mayor sensibilidad al ruido de sistema, pero no lo hemos investigado puesto que no parece tener relación con el uso de contenedores.

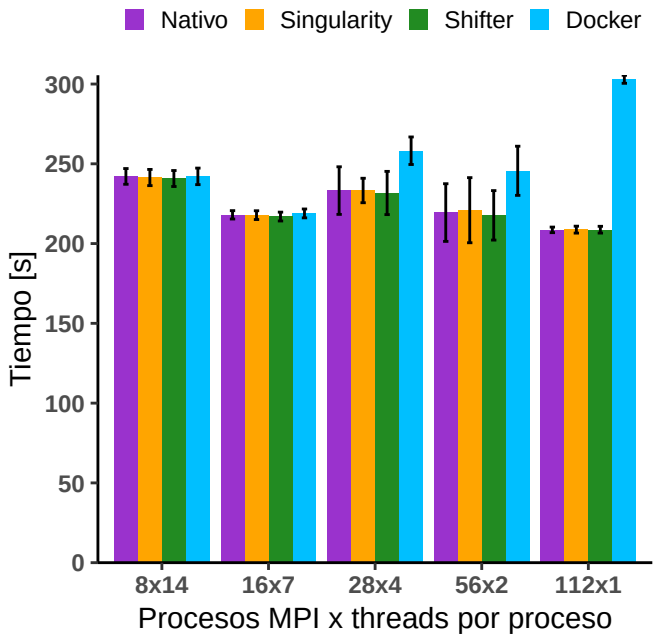
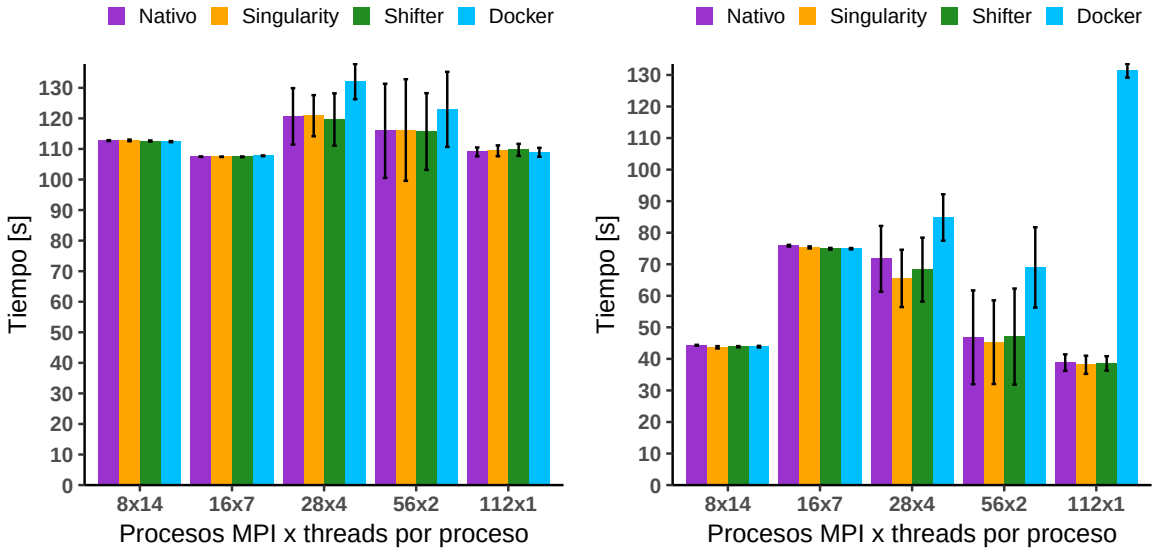


Figura 6.4: Tiempo transcurrido medio de Alya (caso CFD) en Lenox.

Para analizar mejor estos tiempos, vamos a descomponer el tiempo transcurrido en las 2 fases de la simulación. En la Figura 6.5a aparece el tiempo que ha tomado la fase de ensamblaje y en la Figura 6.5b el tiempo del solver.

La fase de ensamblaje es intensa en cálculo y no posee comunicaciones MPI. Podemos ver en la Figura 6.5a como Singularity y Shifter no incurren en costes significantes en esta fase. Con Docker, notamos unos tiempos ligeramente mayores en las distribuciones 28x4 y 56x2, aún así, esto es debido a la variabilidad de estos casos tal como hemos comentado antes. Por lo tanto, ninguna tecnología de contenedor empeora el rendimiento de Alya en una fase de cálculo intenso.

Inspeccionando la duración de la fase del solver en la Figura 6.5b resulta evidente qué causa la degradación de rendimiento con Docker. La fase del solver estresa la red pues es donde se comunican los procesos MPI con llamadas colectivas. Como recordatorio de la subsección 6.2.1,



(a) Tiempo medio de la fase de ensamblaje.

(b) Tiempo medio de la fase de solver.

Figura 6.5: Tiempos de las fases de ensamblaje y solver de Alya (caso CFD) en Lenox.

los contenedores de Docker usan una red virtual para comunicarse, por lo tanto tiene sentido que al aumentar el número de procesos MPI estemos saturando esta red. A diferencia de la versión nativa, Singularity o Shifter, los paquetes de datos que se intercambian los procesos de Docker pasan por esta red virtual y deben ser enrutados mediante Etcd.

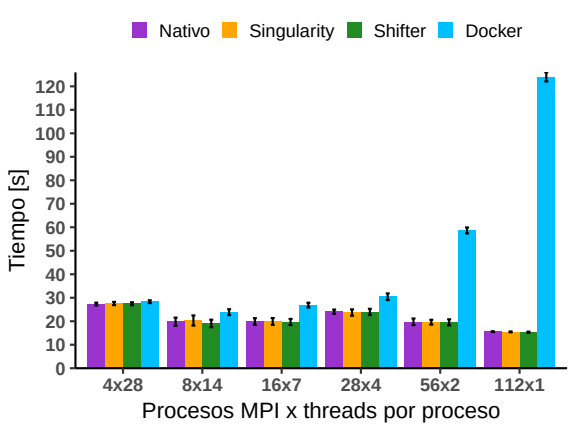
6.3.2. Resultados de BT-MZ

Después de recolectar y procesar las métricas de 50 ejecuciones con cada tecnología, hemos representado los resultados en la Figura 6.6a y 6.6b. Ambas figuras tienen el mismo eje de abscisas, el cual muestra distintas distribuciones de procesos MPI y hilos por proceso. Ambas figuras también muestran vía segmentos la desviación estándar de las medidas tomadas, aunque en la Figura 6.6a el eje de ordenadas contiene tiempo de ejecución en segundos, mientras que la Figura 6.6b muestra GFLOP/s (10^9 operaciones de coma flotante por segundo).

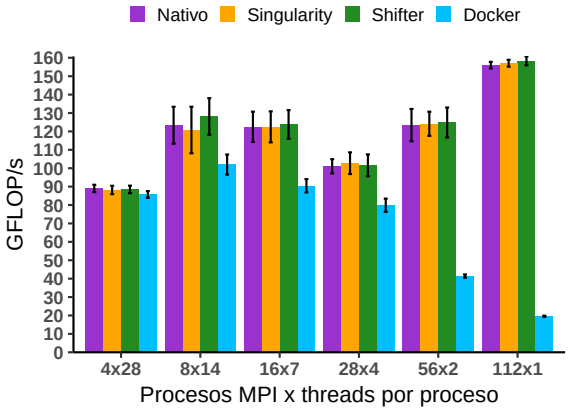
Gracias a estas 2 figuras podemos apreciar los rendimientos de las 3 tecnologías de contenedores. La diferencia de tiempo de ejecución reportada por la Figura 6.6a entre las ejecuciones nativas, Singularity y Shifter con todas las distribuciones es mínima. Ésto lo corrobora de nuevo la Figura 6.6b con los GFLOP/s. Por el contrario, en ambas figuras es evidente como Docker pierde rendimiento a medida que escalamos en procesos MPI.

La degradación de rendimiento de Docker es causada por las comunicaciones MPI tal como hemos podido comprobar al activar las métricas detalladas de BT-MZ. En las Tablas 6.4 se presenta el *profiling* [10] de una ejecución de BT-MZ con varias distribuciones de procesos y tecnologías. La Subtabla 6.4b muestra el tiempo en segundos que se invierte en las fases de CPU y comunicaciones MPI, mientras que la Subtabla 6.4a muestra el porcentaje de uso. Tal como se ve en la Subtabla 6.4b, el tiempo que BT-MZ gasta con Docker en comunicaciones MPI tiende

[10] Esto es, el análisis exhaustivo de la ejecución de un programa midiendo el uso de sus recursos.



(a) Tiempo medio de ejecución.



(b) Media de operaciones de coma flotante por segundo.

Figura 6.6: Rendimiento de BT-MZ con ejecuciones nativas y contenedores.

Distribución \ Fase	Nativo		Singularity		Shifter		Docker	
	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm
4x28	54,10	46,90	54,70	45,30	53,50	46,50	53,00	47,00
8x14	41,70	58,30	41,20	58,80	43,20	56,80	34,60	65,40
16x7	35,50	64,50	35,50	64,50	36,00	64,00	26,20	73,80
28x4	27,60	72,40	27,80	72,20	27,60	72,40	21,70	78,30
56x2	30,80	69,20	31,00	69,00	31,20	68,80	10,40	89,60
112x1	36,60	63,40	36,90	63,10	37,20	62,80	5,20	94,80

(a) Porcentaje que BT-MZ invierte en CPU y comunicaciones MPI.

Distribución \ Fase	Nativo		Singularity		Shifter		Docker	
	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm
4x28	14,73	12,81	15,06	12,49	14,68	12,78	15,00	13,31
8x14	8,26	11,61	8,36	12,04	8,23	10,88	8,25	15,65
16x7	7,06	12,88	7,06	12,89	7,08	12,64	7,03	19,84
28x4	6,63	17,43	6,58	17,13	6,61	17,37	6,60	23,86
56x2	6,08	13,72	6,97	13,55	6,09	13,45	6,09	52,59
112x1	5,69	9,86	5,70	9,75	5,70	9,63	6,44	117,68

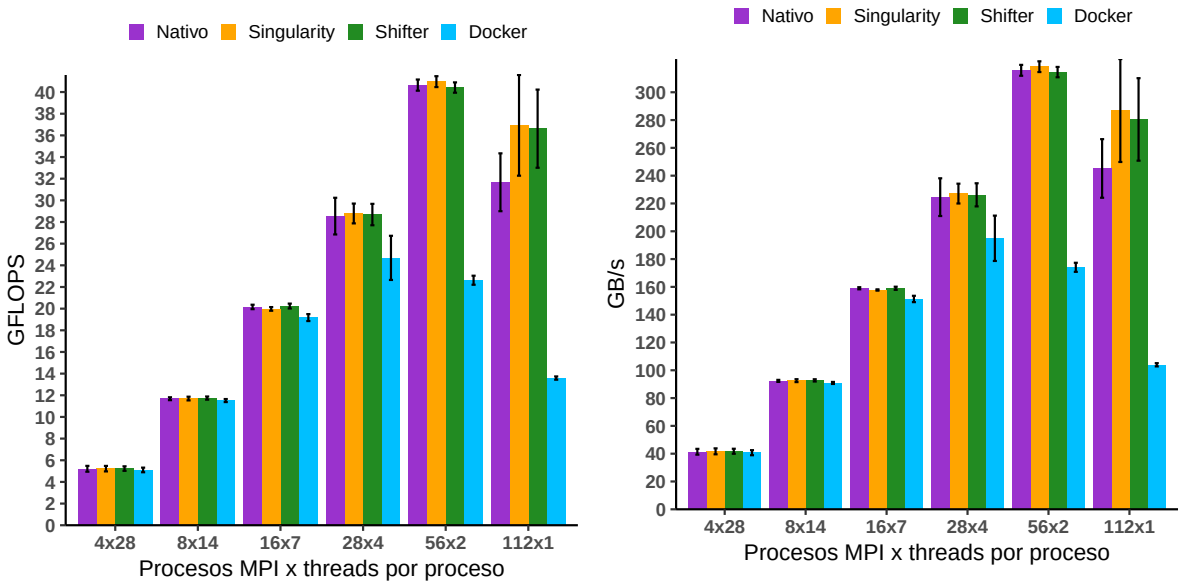
(b) Tiempo (segundos) que BT-MZ invierte en CPU y comunicaciones MPI.

Tabla 6.4: Profiling de BT-MZ con diferentes distribuciones y tecnologías.

a duplicarse junto con la cantidad de procesos. Por ejemplo, de 28x4 a 56x2 el tiempo de comunicación incrementa 28,73 segundos, y de 56x2 a 112x1 crece otra vez 65,09 segundos. Este incremento, sin embargo, no es proporcional al tiempo de CPU, el cual se mantiene similar a las demás tecnologías. En comparación, las métricas de nativo, Singularity y Shifter son muy parecidas y no presentan un incremento tan drástico en comunicaciones MPI. Puesto que la única diferencia entre Docker y las demás tecnologías es el cómo se despliega la aplicación, asumimos que este deterioro del rendimiento es causado por la red virtual de Docker.

6.3.3. Resultados de HPCG

Haciendo la media aritmética de las métricas reportadas por HPCG hemos dibujado la Figura 6.7a y así vemos las diferencias de rendimiento. El gráfico muestra la media de GFLOPS (cuántos más mejor) con 4 métodos de ejecutar el *benchmark*: directamente sobre el anfitrión (nativo), y usando Singularity, Shifter y Docker. Al igual que con BT-MZ, testeamos distintas distribuciones de procesos (eje de abscisas) y mostramos la desviación estándar de cada media con segmentos. Estos resultados validan los obtenidos con BT-MZ, pues se ve como Singularity y Shifter igualan rendimientos nativos mientras que el de Docker se degrada al incrementar progresivamente el número de procesos MPI. El caso 112x1 muestra una ligera anomalía por 2 razones: el rendimiento nativo es significativamente menor que con Singularity y Shifter (aproximadamente 5 GFLOPS, 16,6 % de diferencia relativa respecto Singularity); y la desviación estándar en este caso es notable excepto con Docker.



(a) Operaciones de coma flotante por segundo.

(b) Ancho de banda total contando lecturas y escrituras.

Figura 6.7: Rendimiento de HPCG con ejecuciones nativas y contenedores.

Investigando más detalladamente esta distribución (112x1), hemos visto que el causante de esta diferencia de rendimiento entre nativo y contenedores es el ancho de banda. En la Figura 6.7b se muestra el ancho de banda en GB/s que obtiene cada tecnología. El gráfico de barras resultante resulta idéntico al de la Figura 6.7a. Sin embargo, este ancho de banda reportado por

la aplicación es el total entre memoria y red, por lo que no queda claro la razón exacta de la diferencia.

Examinando por última vez las métricas, nos percatamos que la única función que presenta tiempos de ejecución distintos es la encargada de computar el producto escalar de dos vectores. Justamente esta rutina invoca un `MPI_Allreduce()`, una llamada colectiva de comunicación MPI. Por lo tanto, creemos que la diferencia es debida al ancho de banda de la red. Desafortunadamente, no hemos podido encontrar la razón definitiva porque Lenox se desmanteló durante el proyecto con poco margen de tiempo para reaccionar. Especulamos que el causante es alguna diferencia de librerías entre la distribución de Sistema Operativo de Lenox (Red Hat Enterprise Linux 7) y de los contenedores (Ubuntu 16.04).

6.4. Conclusiones de la Comparativa

En esta sección hemos evaluado el proceso de instalación, el coste de despliegue y rendimiento de 3 implementaciones de contenedores. Del proceso de instalación, hemos concluido que Singularity es el contenedor más simple y seguro de ofrecer en un centro HPC. De la evaluación del coste de despliegue hemos visto como los sobrecostes de Singularity y Shifter son aceptables, aunque una vez más la implementación de Singularity resulta ser la más adecuada por obtener mejores resultados. En este mismo experimento, también hemos concluido que la complejidad de Docker provoca que su coste de despliegue sea inadmisible. Asimismo, el uso de programas de orquestación de contenedores como Kubernetes [11] podría mejorar estos tiempos. No lo hemos investigado pues este estudio sólo trata de evaluar los contenedores pos si mismos.

Finalmente, la evaluación de rendimiento con el caso CFD de Alya, BT-MZ y HPCG ha servido para demostrar y validar la viabilidad de los contenedores. Tanto con aplicaciones científicas reales, como con *benchmarks* que ponen a prueba la capacidad de cálculo y transferencia de memoria, ambos Singularity y Shifter consiguen rendimientos prácticamente idénticos al entorno nativo independientemente del modelo de programación paralelo que se esté explotando (MPI y OpenMP) y su distribución de procesos. Los contenedores de Docker también son capaces de obtener buenos rendimientos, claro que su complejidad de red imposibilita que escalen con procesos MPI. Definitivamente, los contenedores no perjudican la potencia de cálculo de los procesadores, pero dependiendo de su implementación sí pueden requerir capas software extra que ralenticen las aplicaciones debido a la red o operaciones de entrada/salida con dispositivos.

[11] Más información sobre Kubernetes: https://kubernetes.io/

Capítulo 7

Estudio de Portabilidad entre Arquitecturas

Aunque la tecnología de contenedores es ampliamente usada porque mejora la portabilidad del software, ésta sigue estando limitada por atributos hardware específicos. A saber, no es posible ejecutar un contenedor partiendo de una imagen compilada para arquitectura x86 en un procesador PowerPC debido al diferente conjunto de instrucciones de cada arquitectura. La única manera de soportar hardware heterogéneo con contenedores es crear imágenes para cada arquitectura y compilar la aplicación contenida para hacer uso de las características de su anfitrión. Y para incrementar aún más la complejidad, los centros de datos HPC actuales ofrecen varias implementaciones de librerías MPI utilizando la tecnología de red subyacente, por ejemplo, Mellanox Infiniband o Intel Omni-Path. Por lo tanto, un contenedor puede poseer librerías MPI genéricas para ser altamente portable (sacrificando rendimiento), o estar totalmente integrado con la red de alto rendimiento del anfitrión (empeorando su portabilidad).

En este experimento vamos a evaluar la portabilidad de los contenedores con 3 arquitecturas HPC distintas, además de discutir cuál es la mejor manera de desplegar los contenedores para ganar portabilidad o rendimiento. En las secciones a continuación, discutimos las estrategias de despliegue que hemos seguido y mostramos los resultados obtenidos en cada máquina. Finalmente, extraemos las conclusiones de toda la evaluación.

Como ya hemos visto en el capítulo anterior que la cantidad de hilos no afecta al rendimiento, para los siguientes experimentos nos centraremos en la versión puramente MPI de Alya. Como último detalle, remarcamos que todos los experimentos han sido realizados usando los contenedores de Singularity. En las conclusiones del capítulo anterior hemos visto que Singularity es la implementación que mejor satisface los requisito de simplicidad, usabilidad y rendimiento, siendo éstas las razones de por qué descartamos Docker y Shifter.

7.1. Estrategias de Despliegue de Contenedores

Dado que tenemos disponibles varios clústers con distintas arquitecturas y redes (dejamos de lado el clúster Lenox porque su arquitectura queda cubierta por MareNostrum4), hemos in-

geniado 2 estrategias diferentes para desplegar aplicaciones con contenedores. Testeamos estas estrategias con todos los clústers.

- **Singularity genérico**: la imagen del contenedor no soporta ninguna característica específica de su anfitrión. La imagen contiene una instalación genérica de las librerías MPI configurada para usar el protocolo TCP [1] durante las transferencia de datos. Este tipo de contenedor es susceptible de ser ejecutado en cualquier clúster HPC (con la misma arquitectura de procesador) siempre que la versión de las librerías MPI del anfitrión y contenedor sean compatibles porque las llamadas efectuadas por el contenedor deben ser entendidas por el entorno de ejecución MPI de su anfitrión.

- **Singularity integrado**: el contenedor está preparado para soportar la red MPI de alto rendimiento de su anfitrión. Para ello, hemos integrado las librerías MPI del anfitrión en el entorno del contenedor, logrando que el sistema del contenedor sea capaz de usar la tecnología de red óptima, igual que en una ejecución nativa. Esta integración ha consistido en hacer visible todas las librerías y ficheros relacionados con las librerías MPI del anfitrión en el entorno de ejecución del contenedor. De todo el proceso, lo más difícil es localizar los ficheros MPI del anfitrión, ya que una vez son encontrados hacerlos visibles en el contenedor es tan fácil como invocar las opciones que ofrece la línea de comandos de Singularity. Por último, hemos tenido que adaptar las variables de entorno `PATH` y `LD_LIBRARY_PATH` del contenedor para que éste sea capaz de encontrar los nuevos ficheros automáticamente. Siguiendo este método, el contenedor es capaz de invocar el entorno MPI de su anfitrión (el cual presumiblemente está optimizado), pero se sacrifica la portabilidad de la imagen porque ahora está enfocada para un sistema específico.

7.2. Estudio en MareNostrum4

La imagen genérica de Singularity conteniendo Alya es la misma que la usada en los experimentos de Lenox. La imagen integrada, sin embargo, es ligeramente diferente porque modificamos ciertas variables de entorno, además de que ahora el directorio de Alya no se encuentra dentro del sistema de ficheros del contenedor, sino fuera en un directorio de MareNostrum4 ya que debemos compilar la aplicación con el MPI del anfitrión.

La Figura 7.1 muestra el tiempo medio transcurrido de 20 fracciones de tiempo al ejecutar el caso CFD para Alya con 3 versiones. Comparamos el rendimiento de las ejecuciones nativas, de Singularity genérico y Singularity integrado. En el eje de ordenadas se representa el tiempo medio en segundos en escala logarítmica, y en el eje de abscisas la cantidad de nodos usados. El número de procesos MPI que se están usando es igual al número de nodos por 48 cores: $\#procesosMPI = nodos * 48cores/nodo$, es decir, llenamos los nodos completamente. Se nota como el contenedor genérico consigue rendimientos similares a las ejecuciones nativas con una pequeña cantidad de nodos (hasta 8 nodos), pero sufre una importante degradación a medida que escalamos.

Para estudiar el caso con más profundidad, las Figuras 7.2a y 7.2b presentan los tiempos de las fases del ensamblaje y solver respectivamente. En la fase del ensamblaje se ve como todas las versiones consiguen tiempos muy similares, mientras que en el solver destaca lo mal que

[1] El estándar en las comunicaciones del protocolo de Internet.

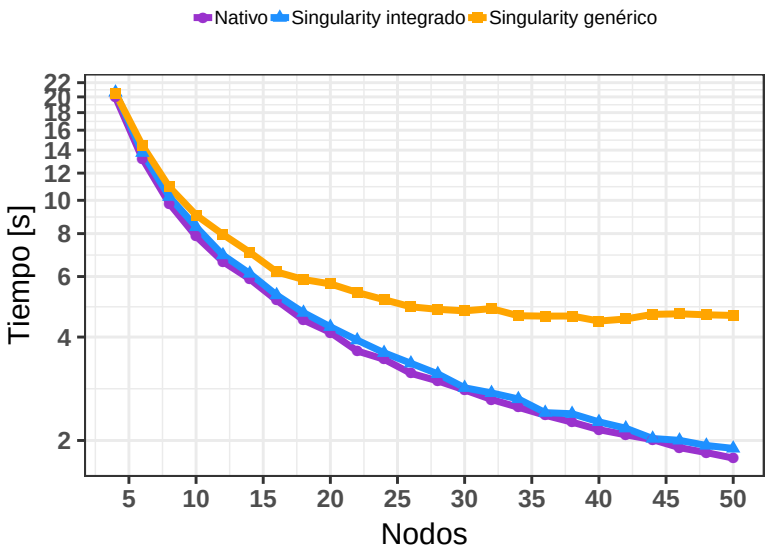
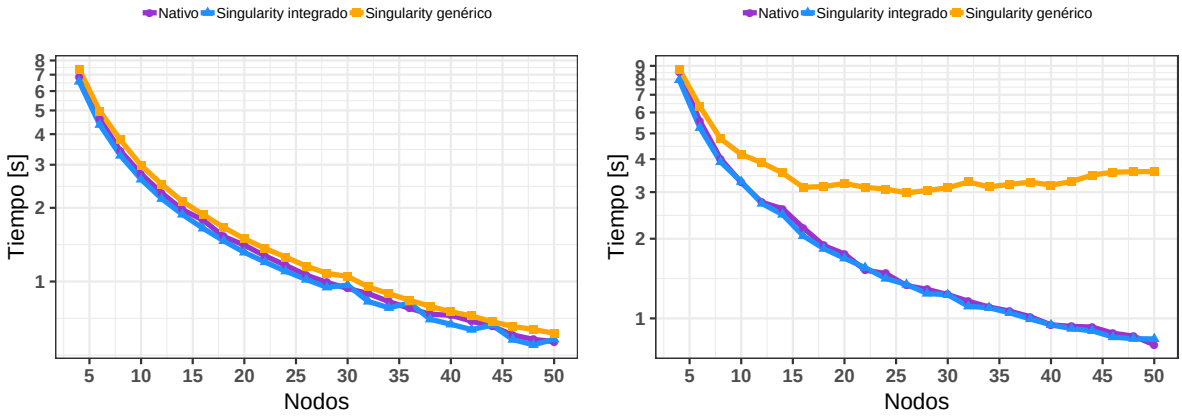


Figura 7.1: Tiempo medio transcurrido del caso CFD de Alya en MareNostrum4.

escala el contenedor genérico. Como Singularity genérico no detecta la red Intel Omni-Path de MareNostrum4, su ejecución de Alya utiliza la red Ethernet del anfitrión, la cuál ofrece un rendimiento muy pobre.



(a) Tiempo medio del ensamblaje.

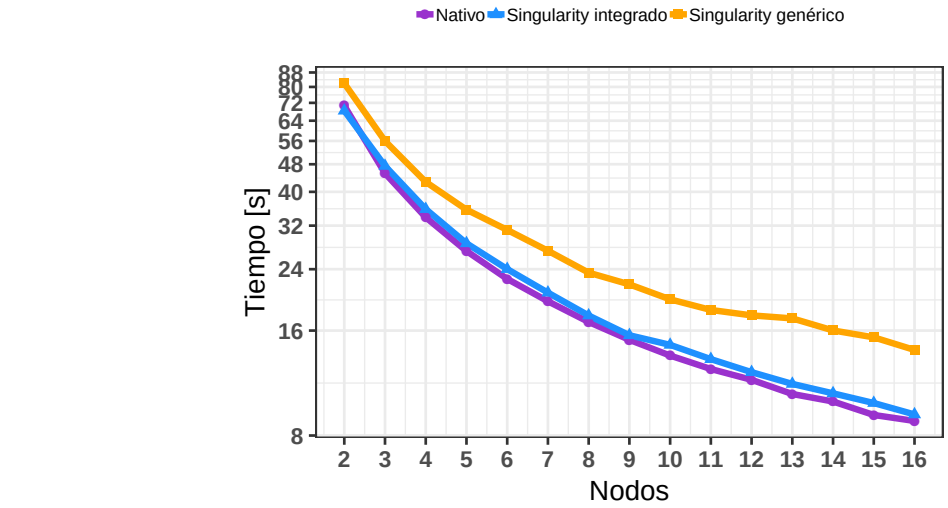
(b) Tiempo medio del solver.

Figura 7.2: Tiempos de las fases del caso CFD en MareNostrum4.

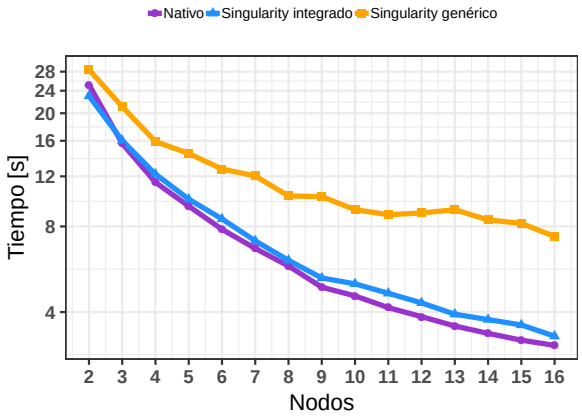
Respecto al tema de la portabilidad, de la manera que hemos creado la imagen integrada ésta tiene la misma portabilidad que una imagen genérica. Singularity integrado carga en tiempo de ejecución las librerías MPI del anfitrión, pero en caso que éstas no estén dispone una instalación genérica MPI.

7.3. Estudio en CTE-POWER

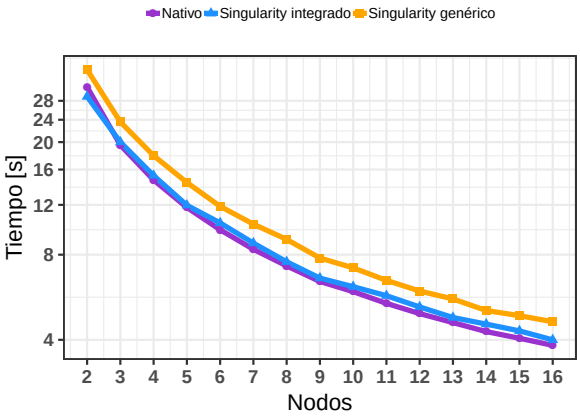
Como la arquitectura de los procesadores de CTE-POWER es incompatible con la de Intel, todas las imágenes que habíamos creado hasta el momento nos han resultado inservibles. Crear las imágenes en CTE-POWER es inviable porque Singularity requiere privilegios de administrador para esta tarea. Como no disponemos de ninguna otra máquina con tal arquitectura, para seguir con este experimento hemos tenido que idear una manera de emular la arquitectura de IBM Power9 (ppc64le) y crear todas las imágenes desde cero. Al final, hemos optado por usar un portátil personal Intel para crear una máquina virtual con QEMU emulando la arquitectura de CTE-POWER. De esta manera, hemos seguido todo nuestro proceso de creación de imágenes adaptándolo a la nueva arquitectura. La parte negativa de esto es que emular hardware es muy costoso, cosa que triplicó el tiempo de creación de las imágenes.



(a) Tiempo transcurrido medio.



(b) Tiempo medio del solver.



(c) Tiempo medio del ensamblaje.

Figura 7.3: Tiempos de las fases del caso CFD en CTE-POWER.

Como la instalación de las librearías MPI de CTE-POWER es distinta de MareNostrum4, hemos integrado de nuevo un contenedor. Hemos identificado los ficheros relacionados con

las instalación MPI y hemos copiado sus referencias al interior del contenedor en tiempo de ejecución. Como resultado, al igual que con MareNostrum4, disponemos de una imagen genérica (que puede ser ejecutada en cualquier otro clúster PowerPC) y una imagen integrada (que reconoce la red Mellanox Infiniband del anfitrión).

Hemos ejecutado de nuevo el caso CFD de Alya en el nuevo sistema desde nativo, Singularity integrado y Singularity genérico. Las Figuras 7.3a, 7.3b y 7.3c muestran los tiempos medios de las fracciones de tiempo completas, de la fase del solver y del ensamblaje respectivamente.

Los resultados muestran el mismo comportamiento que con MareNostrum4. El contenedor integrado prácticamente iguala a las ejecuciones nativas en la fase del solver y ensamblaje. Por el contrario, el contenedor genérico escala muy bien en rendimiento en la fase del ensamblaje, pero está claro que su rendimiento empeora a medida que aumentamos la cantidad de nodos debido a la fase del solver. La red Mellanox Infiniband es un componente hardware específico de CTE-POWER que el contenedor genérico es incapaz de usar.

7.4. Estudio en ThunderX

En ThunderX hemos sufrido la misma problemática que con CTE-POWER. La diferencia de arquitectura del procesador supone una barrera para las imágenes de contenedores, pues estos no abstraen la capa hardware. Al no disponer de privilegios de administrador en ThunderX, hemos creado una máquina virtual con QEMU emulando esta vez la arquitectura Armv8-a (aarch64). Dentro de esta máquina virtual hemos seguido todo el proceso de creación hasta obtener los contenedores finales.

El entorno software de ThunderX es distinto a MareNostrum4 y CTE-POWER, significando que la instalación de las liberías MPI también es diferente. Por ello, hemos realizado el proceso de integración con un contenedor para poder usar el MPI optimizado del anfitrión.

La Figura 7.4 muestra el tiempo medio de una fracción de tiempo en ThunderX al ejecutar Alya. Este último resultado es especialmente interesante porque todas las tecnologías consiguen exactamente el mismo rendimiento con cualquier cantidad de procesos. Ya tenemos comprobado de las ejecuciones anteriores que el contenedor integrado iguala los tiempos de ejecución nativos, que en este caso no es una novedad, pero ahora el contenedor genérico también lo logra. La diferencia de ThunderX respecto a los otros sistemas, es

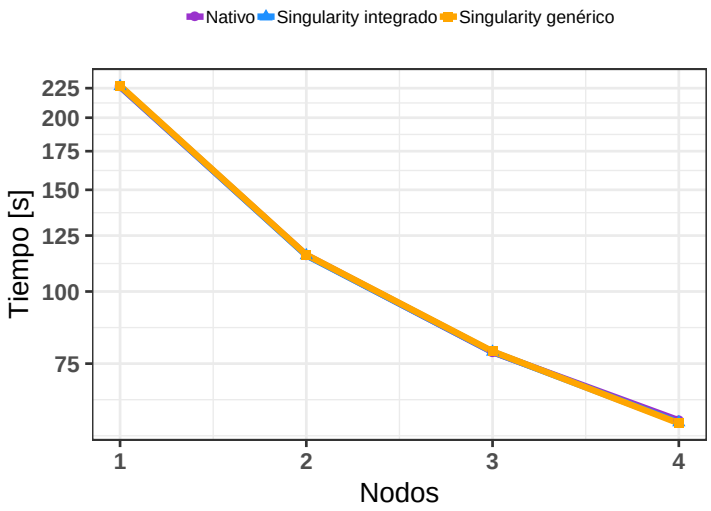


Figura 7.4: Tiempo medio transcurrido del caso CFD en ThunderX.

que este anfitrión no posee una red de alto rendimiento específica, por lo que todas las tecnologías usan la misma red. Debido a que todos usan los mismos componentes hardware y que los contenedores no aportan costes de virtualización, todos los rendimientos son nativos en cualquier caso.

7.5. Conclusiones del Estudio de Portabilidad

Este estudio ha ofrecido valiosa información acerca de la portabilidad de los contenedores y su rendimiento. Para empezar, hemos identificado varias maneras de desplegar los contenedores. Un usuario es capaz de ignorar las características del sistema anfitrión y utilizar un contenedor genérico ganando portabilidad, pero perdiendo rendimiento. O bien, puede dedicar un poco más de tiempo en intentar que su contenedor soporte el hardware específico disponible, logrando así rendimientos nativos, pero con el riesgo de hacer inservible ese contenedor en otros sistemas. Sea como sea, hay un problema imposible de resolver actualmente: la incompatibilidad de las imágenes de contenedor entre diferentes arquitecturas de procesador. Si un usuario dispone de una imagen compilada para Intel, y la desea ejecutar en Arm o PowerPC, no tiene otro remedio que crearla de cero adaptándola a la arquitectura destino.

Queremos hacer énfasis en que la tecnología de contenedores no añade costes de virtualización a la ejecución de aplicaciones. Los resultados muestran claramente como una aplicación HPC real logra rendimientos nativos desde un contenedor, especialmente en el caso de ThunderX. Este último caso es una prueba evidente de que el rendimiento máximo de los contenedores no está limitado por sobrecostes de virtualización, sino por el hardware específico que posea el anfitrión y cómo de preparado esté el entorno software del contenedor para usarlo. Claro está que esta falta de integración no tiene por qué repercutir en pérdidas de rendimiento. Con nuestro caso de uso (Alya) ha quedado retratado como con pocos nodos el rendimiento obtenido se acerca al nativo. Por lo tanto, la diferencia de tiempos entre ejecuciones nativas y genéricas viene dada por la cantidad de uso de un recurso hardware específico: con pocos nodos, la ejecución necesita menos comunicaciones entre nodos y usa menos la red; pero a medida que aumentan la cantidad de nodos, la aplicación deviene limitada por red.

A pesar de las limitaciones de los contenedores causadas por el hardware, creemos que son una solución factible para HPC. La mayoría de centros HPC poseen clústers con arquitectura Intel, por lo que una gran porción de imágenes de los usuarios podrán ser utilizadas. Es cierto que existe el riesgo de perder potencia de cálculo en caso que un contenedor no esté integrado, pero para ello damos la siguiente solución. Los administradores de sistemas de cada centro HPC podrían crear imágenes de contenedores base ya optimizadas para cada clúster, y hacerlas públicas a los usuarios. De esta manera, en caso que un usuario desee portar una aplicación a un clúster nuevo, podría usar esta imagen base para instalar su aplicación encima. En consecuencia, el contenedor resultante aseguraría rendimientos nativos en ese anfitrión. Esta manera de trabajar puede hacer pensar que los contenedores no resuelven los problemas de portabilidad, porque siempre tendremos que estar creando imágenes nuevas para cada clúster. Sin embargo, esto no es totalmente cierto. Con esta metodología se le está ofreciendo al usuario la libertad y comodidad de portar aplicaciones des de su ordenador personal y con la posibilidad de automatizar la creación de la imagen mediante *scripts* o recetas.

Capítulo 8

Test de Escalabilidad

Nuestra última evaluación de los contenedores consiste en un test de escalabilidad fuerte [1]. La inmensa potencia de cálculo que requieren las aplicaciones HPC nos obliga a ejecutarlas con centenares o miles de procesos paralelos. Hasta ahora ya hemos presentado resultados ejecutando Alya con hasta 50 nodos de cálculo (2.400 cores), pero este valor no es suficiente para comprobar si los contenedores tienen alguna limitación a la hora de escalar. Por ello, vamos a ejecutar el caso fluido-estructural de Alya en el supercomputador MareNostrum4 hasta 256 nodos (12.288 cores), una cantidad de recursos más adecuada para este fin.

En la Figura 8.1 se presenta los resultados de este test. La gráfica muestra la ganancia ideal (líneas discontinuas, calculada como: $Ideal = nodos\_usados/4$), la ganancia nativa, con el contenedor integrado y el genérico respecto 4 nodos hasta 256, ejecutando 1 proceso MPI por core (por lo que ejecutamos la versión sólo MPI de Alya). Una vez más, podemos determinar que el contenedor integrado logra las mismas ganancias que las ejecuciones nativas aún escalando hasta decenas de miles de cores. Parecido a como hemos visto en el test de portabilidad, el contenedor genérico consigue buenas ganancias con poca cantidad de recursos, pero a partir de 32 nodos empieza a perder rendimiento. De hecho, hemos sido incapaces de ejecutar este contenedor con 256 nodos porque la red Ethernet no es suficientemente rápida para gestionar todas las comunicaciones

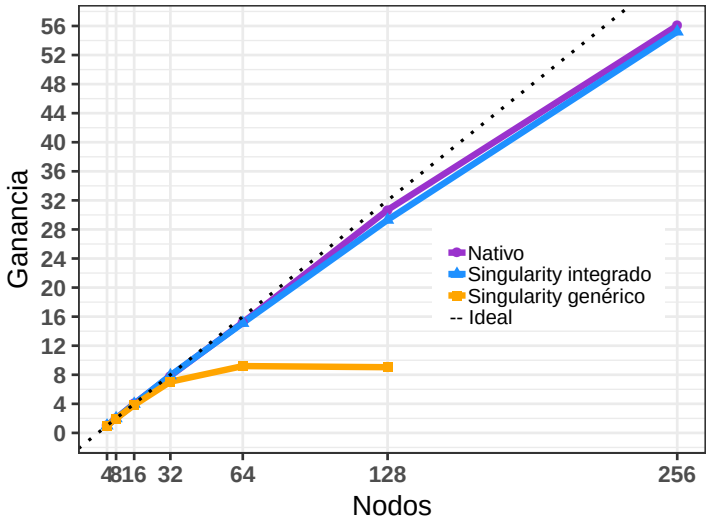


Figura 8.1: Ganancias de la escalabilidad fuerte del caso FSI de Alya en MareNostrum4 con 3 métodos de ejecución.

[1] Esto es, el tamaño del problema que se resuelve es constante, pero la cantidad de recursos que se usan se incrementa.

MPI de la fase del solver, causando que la ejecución se suspenda.

Capítulo 9

Planificación

En este capítulo se presenta la planificación del proyecto. Este proyecto se inició dentro de la empresa BSC en julio de 2018 y su finalización está prevista para finales de junio de 2019. La duración total del proyecto es, aproximadamente, 12 meses.

A continuación, se detalla la planificación inicial de este proyecto junto con las potenciales alteraciones o imprevistos que se detectaron cuando empezó. También se describen las tareas iniciales que se definieron y se mostrará un inventario de los recursos necesarios previstos.

Después de presentar la planificación inicial, se mostrará la planificación final resultante. En esta sección se explicarán los problemas acontecidos durante la realización del proyecto, como se han solventado y su repercusión en el proyecto.

9.1. Planificación Inicial

Esta sección expone la planificación realizada al iniciarse el proyecto. A continuación se presentan las tareas planificadas para este proyecto. Las tareas se muestran siguiendo el orden cronológico deseado para su ejecución. Al final de la sección se muestra un diagrama de Gantt con la planificación de las tareas en un calendario.

9.1.1. Especificación de Tareas

Es importante diferenciar 2 tipos de tareas: las tareas básicas destinadas a avanzar con el proyecto, y las tareas de control destinadas a supervisar la correcta ejecución del trabajo. Las tareas básicas se identifican de la forma "T#" mientras que las tareas de control de la forma "TC#".

T1 - Gestión del proyecto. Tiempo estimado: 75h

Al empezar es necesario definir los detalles del proyecto. Respecto al trabajo que se realizará, hay que contextualizar el proyecto y definir sus objetivos, agentes implicados y metodología.

Para asegurar que el proyecto progresa adecuadamente, también hay que planificar las tareas que se ejecutarán y sus recursos. Una vez los objetivos y las tareas queden claros, sólo quedará preparar el presupuesto del proyecto y valorar la sostenibilidad del mismo.

T2 - Estudiar las tecnologías de contenedores. Tiempo estimado: 80h

Como todo el proyecto gira entorno a la tecnología de virtualización a nivel de sistema operativo, resulta imprescindible estudiar los principios de su funcionamiento. Este estudio involucra repasar los distintos tipos de virtualización existentes, aparte de los contenedores mismos para comprender la motivación de su desarrollo y conocer sus características. Además del estudio de la tecnología en general, aquí también se incluye el estudio de las 3 implementaciones que se van a evaluar: Docker, Singularity y Shifter. Estudiar las implementaciones de contenedores conlleva 2 acciones:

1. Aprender a usar la interfaz de usuario de cada implementación.

2. Identificar en qué se caracteriza cada implementación, sus puntos débiles y fuertes.

T3 - Aprender a usar los entornos de prueba. Tiempo estimado: 40h

Estudiar los contenedores es fundamental al ser el objeto de estudio del proyecto, sin embargo, es imposible que el trabajo se desarrolle satisfactoriamente sin antes familiarizarse con los clústers donde se van a realizar las pruebas. Está previsto realizar experimentos con contenedores en las siguientes máquinas: el supercomputador MareNostrum4, los clústers CTE-POWER y CTE-KNL del BSC, el clúster Lenox de Lenovo y el clúster ThunderX del proyecto Mont-Blanc. Todas las máquinas ofrecen unas prestaciones tanto hardware como software distintas, por lo tanto los objetivos de esta tarea son:

- Obtener acceso a las máquinas: conseguir un usuario y conectarse a ellas

- Identificar de qué hardware dispone la máquina: fabricante, modelo y tipo de arquitectura del procesador, cantidad de memoria y tipo de red.

- Identificar el SO que usa la máquina y el software disponible: distribución del SO, versión del kernel de Linux, versiones de la librerías MPI disponibles y versión del gestor de colas SLURM.

- Aprender a usar la interfaz de usuario de las librerías MPI

- Aprender a enviar *scripts* mediante el gestor de colas para su ejecución sin interferencias de procesos ajenos en los nodos de la máquina

Es realmente importante destacar la necesidad de aprender a usar correctamente la interfaz de MPI y el gestor de colas de cada máquina, ya que de ésto depende la correcta ejecución de los experimentos. El entorno MPI y el gestor de colas va a permitir a definir exactamente la cantidad de recursos hardware destinados a cada experimento y asegurar que las ejecuciones no sufren interferencias por culpa de otros usuarios de la máquina.

T4 - Diseñar los experimentos. Tiempo estimado: 20h

Para obtener resultados significativos es esencial discutir con el director, codirectora e in-

vestigador del proyecto los detalles de los experimentos a llevar a cabo. Esto es: definir la configuración de recursos MPI-OpenMP con la que se van a ejecutar las aplicaciones, decidir la cantidad de muestras a sacar de cada experimento y definir qué versiones de MPI, compilador de GNU, *benchmarks*, Docker, Singularity y Shifter se van a usar.

T5 - Instalar Docker, Singularity y Shifter. Tiempo estimado: 40h

Debido a que los clústers HPC que se van a usar no disponen por defecto del software de contenedores, será necesario realizar instalaciones manuales, o bien solicitar al equipo de administradores de sistemas de cada máquina la instalación de las aplicaciones. En el caso concreto de Lenox, habrá que instalar manualmente Docker, Singularity y Shifter. En el caso de MareNostrum4, CTE-POWER, CTE-KNL y ThunderX habrá que solicitar la instalación de los contenedores a los administradores de sistemas de cada máquina porque son computadores en producción y la instalación requiere derechos de administrador. Por último, habrá que verificar que todos los contenedores funcionan correctamente después de las instalaciones.

T6 - Pedir acceso al código de Alya. Tiempo estimado: 4h

En la sección de la introducción de este proyecto se ha mencionado que usaremos una aplicación CFD de simulación biológica en producción para probar los contenedores. Esta aplicación se llama Alya, un simulador desarrollado dentro del BSC el cual tiene licencia privada, por lo que será necesario solicitar acceso al código al departamento de CASE (Computer Applications in Science and Engineering).

T7 - Instalar Alya, BT-MZ y HPCG en los clústers. Tiempo estimado: 28h

Alya, BT-MZ y HPCG son las 3 aplicaciones que se van a usar en el proyecto para evaluar los contenedores. Es necesario instalar las aplicaciones manualmente en todas las máquinas para poder ejecutar los experimentos sin usar contenedores.

T8 - Crear imágenes de Docker, Singularity y Shifter con Alya, BT-MZ y HPCG. Tiempo estimado: 40h

Para realizar la evaluación de los contenedores es necesario crear imágenes que contengan todo el entorno virtualizado de Alya, BT-MZ y HPCG.

T9 - Realizar la comparativa entre Docker, Singularity y Shifter. Tiempo estimado: 80h

Ejecutar las aplicaciones Alya, BT-MZ y HPCG con y sin contenedores en Lenox, recoger las métricas, procesar los resultados y validarlos.

T10 - Realizar el experimento de portabilidad con Alya. Tiempo estimado: 80h

En caso necesario, adaptar las imágenes de Singularity para poder ejecutar Alya en MareNostrum4, CTE-KNL, CTE-POWER y ThunderX. A continuación, preparar las ejecuciones, recoger sus métricas, procesar los resultados y validarlos.

T11 - Realizar el test de escalabilidad de Alya. Tiempo estimado: 80h

El último experimento consiste en realizar un test de escalabilidad fuerte de Alya en el supercomputador MareNostrum4 (hasta 12 mil cores). Habrá que recoger los resultados, pro-

cesarlos y validarlos.

T12 - Redactar la memoria del proyecto. Tiempo estimado: 60h

La tarea consiste en acordar una tabla de contenidos con los directores y redactar todo el proyecto con su respectiva introducción, metodología, resultados, conclusiones, etc.

T13 - Preparar la defensa del TFG. Tiempo estimado 16h

Para la defensa del TFG será necesario preparar la presentación oral.

TC1- Reuniones con el director y codirector. Tiempo estimado 44h

Bisemanalmente se reunirán el director, codirector e investigador para asegurar que el proyecto progresa adecuadamente, detectar desvíos o obstáculos y verificar que los resultados obtenidos hasta la fecha son correctos.

TC2 - Prevención de alteraciones. Tiempo estimado: 0-240h

Debido a que la planificación inicial muestra que potencialmente sobrará tiempo, esta tarea de control está dedicada a suplir posibles alteraciones del proyecto. Por ejemplo, que una máquina se estropee y no pueda usarse durante X tiempo.

En la Tabla 9.1 se muestra un resumen completo de todas la tareas planificadas. En el cuadro se especifica las horas estimadas para realizar cada tarea así como sus dependencias y recursos imprescindibles. En total y dependiendo del tiempo de prevención que sea necesario consumir, el proyecto tomará entre 687 y 927 horas.

9.1.2. Posibles Desviaciones o Alteraciones

Este proyecto contiene tareas de todo tipo. Por un lado, están las tareas T1, T2, T3, T4, T6, T12 y T13, todas caracterizadas por su simplicidad y porque los recursos de los que dependen tienen una muy baja probabilidad de fallo, al menos de manera grave para el proyecto. Por otro lado, las tareas T5, T7, T8, T9, T10 y T11 presentan un cierto nivel de incertidumbre y riesgo implícito, ya que a priori no se sabe si el software es compatible con todas las máquinas HPC ni si éstas van a sufrir averías o decomisiones (ya que algunos clústers, como Lenox, son experimentales).

Mientras que el primer grupo de tareas involucra solamente investigar lo ya existente, es improbable que aparezcan desviaciones ya que el entorno es poco variable. El segundo grupo, sin embargo, incluye realizar instalaciones y experimentos que nadie antes ha realizado en los sistemas HPC del proyecto. Como estas tareas se adentran en lo desconocido, se pueden estimar alteraciones.

En el peor de los casos, se contemplan varias posibilidades:

- Las máquinas HPC dejan de funcionar por una avería o decomisión.
- Instalar el software necesario resulta inviable.

Tarea	Descripción	Tiempo est. (horas)	Dependencias	Recursos imprescindibles
T1	Gestión del proyecto	75	-	Director, codirector, investigador
T2	Estudio de los contenedores	80	T1	Investigador
T3	Estudio entorno de pruebas	40	T1	Investigador, máquinas HPC
T4	Diseño de los experimentos	20	T1, T2, T3	Director, codirector, investigador
T5	Instalación de contenedores	40	T1, T2, T3	Investigador, máquinas HPC, sys-admins
T6	Solicitar acceso a Alya	4	T1	Director, desarrollador jefe de Alya
T7	Instalar Alya, BT-MZ y HPCG	28	T1, T3, T6	Investigador, desarollador de Alya, máquinas HPC
T8	Crear imágenes de contenedores	40	T1, T2, T3, T4,T5, T6	Investigador
T9	Comparativa de contenedores	80	T1, T2, T3, T4, T5, T6, T7	Investigador, máquinas HPC
T10	Evaluación de portabilidad	80	T1, T2, T3, T4,T5, T6, T7	Investigador, máquinas HPC
T11	Test de escalabilidad	80	T1, T2, T3, T4, T5, T6, T7	Investigador, máquinas HPC
T12	Redacción de la memoria	60	T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11	Director, codirector, investigador
T13	Preparación de la defensa	16	T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11	Director, codirector, investigador
TC1	Reuniones con el equipo	44	-	Director, codirector, investigador
TC2	Prevención	0-240	-	-

Tabla 9.1: Tabla resumen de las tareas.

- Los resultados de los diferentes experimentos resultan ininteligibles.

En caso que las máquinas HPC sufrieran una avería dependería de la gravedad de ésta. Si es algo temporal, el proyecto debería poder adecuarse sin problemas ya que tenemos un margen de prevención. Si la avería resulta fatal o sufre un desmantelamiento, sin embargo, se tendría que prescindir de la máquina en cuestión sin ninguna alternativa. Por ello, es importante realizar todos los experimentos de manera efectiva y rápida para evitar este posible riesgo, así como pensar un plan de contingencia por si el problema se materializa.

Si la instalación de algún programa (un contenedor o *benchmark*) resultara inviable existen varias alternativas. Una sería modificar el código fuente del programa para adaptarlo al computador en específico. En caso que esta alternativa no resultara factible, ya sea porque depende de una cuestión hardware, o porque es demasiado trabajo, se podría mirar de reemplazar el contenedor o *benchmark* por algún otro. En última instancia, se podría omitir el uso de algún contenedor o *benchmark* problemático.

En caso que se detecte que los resultados de algunos experimentos no tienen sentido, se procedería a revisar las salidas de las ejecuciones y consultar el estado de la máquina a los administradores. De una manera u otra, se tendría que acabar descubriendo la causa del fallo y por tanto solucionarlo.

9.1.3. Recursos Necesarios

Los recursos necesarios para ejecutar las tareas descritas hasta ahora consisten en la siguiente lista.

- Material informático (ordenador, ratón, teclado, etc.)
- Conexión a internet
- Un lugar de trabajo (oficina)
- Acceso a MareNostrum4, CTE-POWER, CTE-KNL, Lenox y ThunderX
- Acceso a Docker, Singularity y Shifter
- Acceso a Alya, BT-MZ y HPCG
- Una herramienta de análisis estadístico (R [1], por ejemplo)
- Un editor de textos (Overleaf)
- Una lista de contactos de los administradores de sistemas de las máquinas HPC y el jefe desarrollador de Alya
- Una sala de reuniones

[1] R es un lenguaje de programación enfocado al análisis estadístico. Más información en: https://www.r-project.org/

9.1.4. Diagrama de Gantt

En la Figura 9.2 se muestra la planificación temporal de las tareas en un calendario mediante un diagrama de Gantt. Aunque en el diagrama no está explícito, bisemanalmente se celebrará una reunión de aproximadamente 4 horas de acuerdo con la tarea TC1.

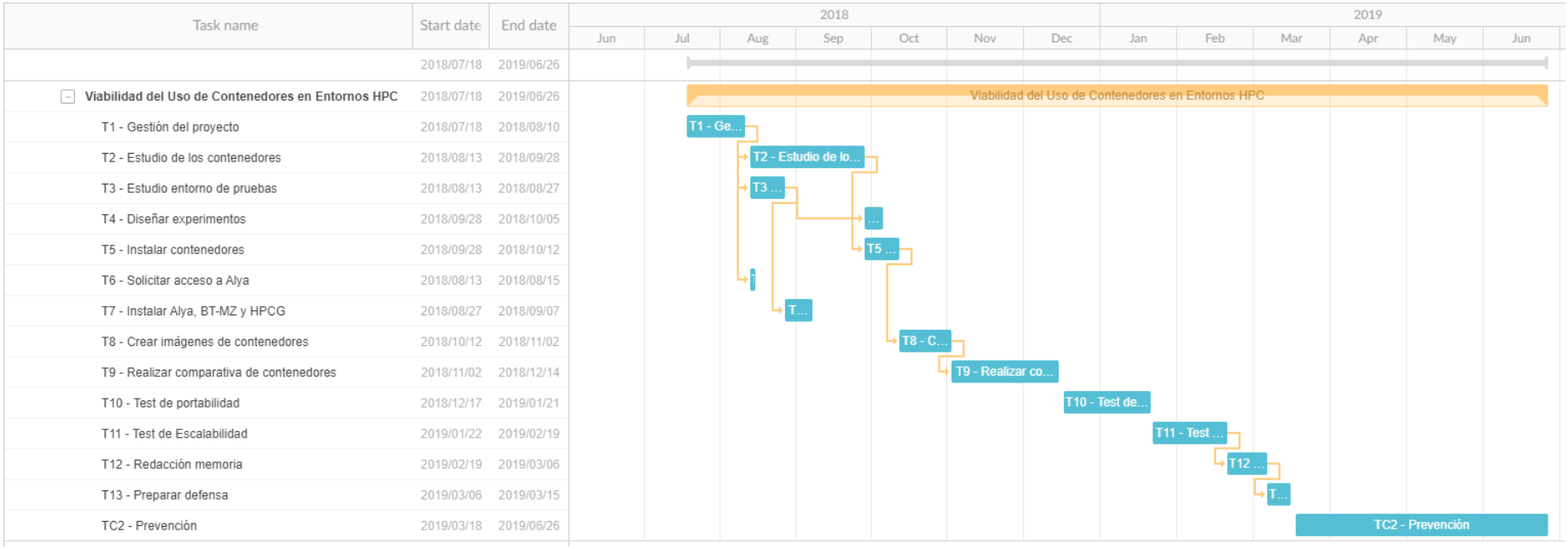


Figura 9.1: Diagrama de Gantt de la planificación inicial.

9.2. Planificación Final

En esta sección se presenta la planificación final del proyecto, resultado de las alteraciones que han ido aconteciendo. En la Tabla 9.2 se muestra un resumen del estado de todas las tareas. Se puede apreciar en la tabla como el proyecto se ha desarrollado bastante afín a la planificación inicial. Aún así, ciertas tareas han sufrido desviaciones (sucesos que no han influido en el resultado final) o imprevistos (sucesos que han obligado a modificar los objetivos del proyecto) que han obligado a modificar levemente la planificación inicial del proyecto.

Tarea	Descripción	Estado	Observaciones
T1	Gestión del Proyecto	CE	Ningún imprevisto.
T2	Estudio de los contenedores	CE	Ningún imprevisto.
T3	Estudio de los entornos de pruebas	CE	Ningún imprevisto.
T4	Diseño de los experimentos	CE	Ningún imprevisto.
T5	Instalación de contenedores	CD	La instalación de Shifter en la máquina Lenox fue muy compleja. Singularity no funciona en CTE-KNL por algún problema con sus procesadores.
T6	Solicitar acceso a Alya	CE	Ningún imprevisto.
T7	Instalar Alya, BT-MZ y HPCG	CE	Ningún imprevisto.
T8	Crear imágenes de contenedores	CI	Para CTE-POWER y ThunderX se tuvo que idear una manera de crear contenedores para cada arquitectura de procesador.
T9	Comparativa de contenedores	CI	El clúster Lenox ha sido desmantelado antes de poder terminar con todas las pruebas.
T10	Evaluación de portabilidad	CE	Se complicó el proceso de integrar los contenedores con la instalación MPI de cada host.
T11	Test de escalabilidad	CI	No fue posible ejecutar toda la escalabilidad con un tipo de contenedor.
T12	Redacción de la memoria	CE	Ningún imprevisto.
T13	Preparación de la defensa	P	La tarea se iniciará una vez la memoria sea publicada.

Tabla 9.2: Tabla resumen de tareas. **CE**: completada con éxito; **CD**: completada con desviaciones; **CI**: completada con imprevistos; **EP**: en progreso; **P**: pendiente.

Repasando en orden las tareas que han dado problemas:

- **T5** (Instalación de los contenedores): Por un lado, la instalación de Shifter en la máquina Lenox resultó ser más complicada de lo esperado. Se tuvo que invertir un poco más de tiempo del previsto **(12 horas de retraso)** para resolver dependencias, configurar Shifter y los diversos programas de terceros que usa como MongoDB, Munge y gunicorn. Por el otro lado, no se pudo instalar Singularity en CTE-KNL. CTE-KNL es una máquina en producción del BSC, y de su gestión se encargan los administradores de sistemas, por lo que nosotros no tenemos ningún poder a la hora de instalar software que requiera privilegios de administrador. Los administradores de sistemas accedieron a instalar Singularity, pero resultó imposible de hacer funcionar por algún problema con los procesadores Intel Xeon Phi Knights Landing. Al no disponer de otra alternativa viable de contenedores, y porque Intel ha anunciado que abandonará el desarrollo de la arquitectura Knights Landing [2], se ha decidido prescindir de la máquina CTE-KNL.

- **T8** (Crear imágenes de contenedores): Debido a que CTE-POWER y ThunderX poseen un ISA (Instruction Set Architecture) poco habitual, no ha sido posible crear directamente con un portátil convencional las imágenes de contenedores para estas 2 arquitecturas. Como solución, se ha ideado virtualizar las arquitecturas `ppcle64` y `aarch64` mediante la máquina virtual QEMU en un portátil Intel personal. Este hecho ha supuesto un retraso de la tarea **36 horas de retraso)**, pues se ha tenido que crear todo el entorno virtual y la creación de las imágenes es mucho más lenta en un entorno virtualizado.

- **T9** (Comparativa de contenedores): A pesar de haber realizado todas las ejecuciones para hacer la comparativa entre tecnologías, la máquina Lenox fue desmantelada antes de tiempo. Hay un caso concreto a la hora de ejecutar HPCG que da unas métricas de rendimiento un poco raras y que no hemos tenido tiempo de averiguar su porqué. Tampoco era razonable buscar o comprar una nueva máquina y trasladar todo el entorno allí.

- **T10** (Evaluación de portabilidad): La integración de los contenedores con las redes que usan las librerías MPI de MareNostrum4, CTE-POWER y ThunderX resultó ser más compleja de lo esperado. Se logró integrar los contenedores con el sistema huésped después de varias iteraciones, pero con un ligero retraso en la planificación inicial **(28 horas de retraso)**.

- **T11** (Test de escalabilidad): No ha sido posible ejecutar un caso concreto del test con un tipo de contenedor debido a limitaciones hardware. Sin embargo, este imprevisto no afecta al resultado global del experimento.

El diagrama de Gantt resultante de estas desviaciones e imprevistos se muestra en la Figura 9.2

[2] Requiem for a Phi: Knights Landing Discontinued: https://www.hpcwire.com/2018/07/25/end-of-the-road-for-knights-landing-phi/

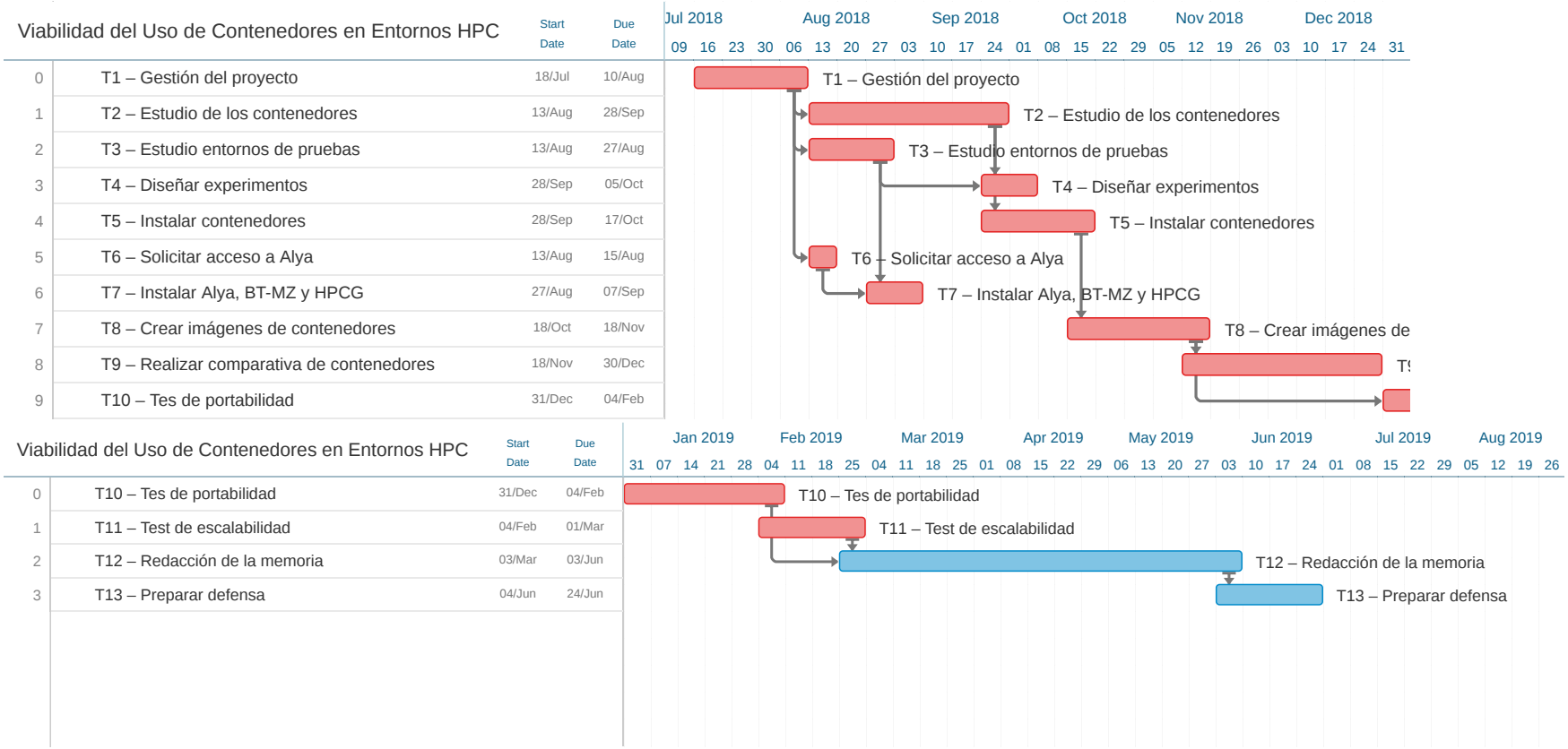


Figura 9.2: Diagrama de Gantt resultante de la ejecución del proyecto.

En definitiva, a pesar de los imprevistos y desviaciones que han ocasionado un ligero retraso y han obligado a prescindir de ciertos experimentos, el trabajo se ha podido desarrollar satisfactoriamente y dentro de lo esperado.

9.2.1. Impacto en los Costes

Estos imprevistos y desviaciones no tienen una repercusión directa en los costes del proyecto. Indirectamente, las horas que han dedicado los administradores de sistemas para instalar Singularity en CTE-KNL no han sido de utilidad, y los diversos retrasos ya estaban contabilizados en el apartado de contingencia del presupuesto inicial.

Capítulo 10

Costes del Proyecto

En este capítulo se presenta el coste aproximado del proyecto considerando todos los recursos que se van a usar. A continuación, se expone el valor de los recursos hardware, software y recursos humanos requeridos así como los costes indirectos del proyecto.

10.1. Recursos Hardware

En la Tabla 10.1 aparecen los recursos hardware que se van a requerir con sus respectivos costes. Para el caso de los clústers Lenox y ThunderX resulta imposible contabilizar el coste de estas máquinas porque la información no está accesible, pero aún así es importante hacer constar su uso en el presupuesto. Asimismo, sólo se sabe el coste total del conjunto de supercomputadores MareNostrum4, CTE-POWER y CTE-KNL del BSC.

Producto	Precio (€)	Vida útil (años)	Amortización (€)
Portátil Dell Latitude 7480	1.400	4	350
Monitor Dell	250	4	62,5
Ratón Logitech	15	4	3,75
Teclado Logitech	20	4	5
MareNostrum4 CTE-POWER CTE-KNL	34.000.000	4	N/A
Lenox	N/A		
ThunderX	N/A		
Total	34.001.685		416,25

Tabla 10.1: Presupuesto de recursos hardware.

10.2. Recursos Software

Dentro del grupo de recursos software destaca la aplicación de simulación CFD Alya, los benchmarks BT-MZ y HPCG, y las implementaciones de contenedores Docker, Singularity y Shifter. Por suerte, Alya ha sido cedida gratuitamente para el proyecto, y tanto los benchmarks como los contenedores están tutelados bajo la licencia GNU GPL, la cual permite su uso gratuitamente. En resumen, todo el software requerido por el proyecto tiene coste 0.

- OpenSuse Leap 42.3
- Editor Latex
- Alya
- *Benchmark* BT-MZ
- *Benchmark* HPCG
- Docker
- Docker Hub
- Singularity
- Shifter

10.3. Recursos Humanos

Dentro de los RRHH se encuentran, de mayor a menor coste para el proyecto: el investigador encargado de realizar los experimentos y evaluar los resultados; el director y codirectora del proyecto los cuáles dirigirán y validarán el trabajo realizado; el desarrollador jefe de Alya quién dará acceso a su código; el desarrollador de Alya quién dará soporte con el uso de la aplicación; y finalmente los administradores de sistemas de las distintas máquinas usadas en este proyecto. Una vez conocido el personal involucrado, en la Tabla 10.3 se expone su coste por hora bruto, el cual ha sido deducido de la página web: `https://www.payscale.com/research/ES/Country=Spain/Salary`.

En la Tabla 10.2 se estiman las horas que dedicará cada rol de RRHH a cada tarea en específico. Finalmente, gracias a la información proporcionada por las Tablas 10.3 y 10.2 se puede contabilizar el coste final de los RRHH tal como está en la Tabla 10.4.

Tarea	Horas	Dedicación (h)					
		Director	Codirec.	Invest.	Admin.	Des. Jefe	Des.
T1-Gestión del Proyecto	75	25	25	25	0	0	0
T2-Estudio de contenedores	80	0	0	80	0	0	0
T3-Estudio de entornos HPC	40	0	0	40	0	0	0
T4-Diseño experimentos	20	7	7	6	0	0	0
T5-Instalación contenedores	40	0	0	40	0	0	0
T6-Solicitud de acceso a Alya	4	2	0	0	0	2	0
T7-Instalación benchmarks	28	0	0	22	0	1	5
T8-Creación imágenes	40	0	0	36	0	0	4
T9-Comparativa contenedores	80	0	0	75	5	0	0
T10-Test de portabilidad	80	0	0	70	10	0	0
T11-Test de escalabilidad	80	0	0	75	5	0	0
T12-Redacción memoria	60	10	10	40	0	0	0
T13-Preparación defensa	16	2	2	12	0	0	0
TC1-Reuniones	44	15	15	14	0	0	0
TC2-Prevención	-	-	-	-	-	-	-
Total	687	61	59	535	20	3	9

Tabla 10.2: Tiempo estimado que cada rol dedicará a cada tarea del proyecto.

Rol	Coste por hora (€/h)
Director	26
Codirectora	26
Ingeniero de investigación junior	11
Administrador de sistemas	13
Desarrollador software jefe	18
Desarrollador software	13

Tabla 10.3: Coste por hora bruto de los RRHH.

Rol	Horas	Coste (€)
Director	61	1.586
Codirectora	59	1.534
Ingeniero de investigación junior	535	5.885
Administrador de sistemas	20	260
Desarrollador software jefe	3	54
Desarrollador software	9	117
Total	687	9.436

Tabla 10.4: Presupuesto de RRHH.

10.4. Costes Indirectos

Como el proyecto se debe realizar en algún lugar dedicado a la tarea, hay que contabilizar los costes de la oficina, su mobiliario, luz, limpieza etc. En la Tabla 10.5 se resumen estos gastos.

10.5. Presupuesto Final

Después de presentar los costes de todos los recursos por separado, es posible presentar el presupuesto final de este proyecto en la Tabla 10.6. A raíz de los imprevistos descritos en la Sección 9.2, el investigador ha tenido que dedicar 76 horas de más en el proyecto ($76h * 11€/h = 836€$). Tanto dirección como los demás agentes implicados han mantenido sus horas de trabajo dentro de la planificación inicia. En la Tabla 10.6 aparece el presupuesto final de este proyecto.

Recurso	Coste (€)
Alquiler de la oficina (12 meses)	5.000
Mobiliario	1.500
Gastos de la oficina (electricidad, agua, seguridad, limpieza, etc.)	3.000
Total	9.500

Tabla 10.5: Presupuesto de costes indirectos.

Concepto	Presupuesto (€)
Recursos Hardware	416,25
Recursos Software	0
Recursos Humanos	9.436
Costes Indirectos	9.500
Total	19.352,25
Imprevistos	836
Total	21.563,25

Tabla 10.6: Presupuesto final.

Este proyecto ha costado aproximadamente 21.563,25 euros.

10.6. Control del Presupuesto

Este proyecto parte de una base económica muy sólida por 2 razones. La primera, todas las máquinas HPC han sido cedidas por el BSC gracias al proyecto de la Comisión Europea HPC-Europa3, el cual tiene una duración mayor que este trabajo. La segunda, todo el software usado es gratuito. Como consecuencia, lo único que podría ocasionar algún tipo de imprevisto económico es que se estropeara el portátil o bien que una tarea del proyecto se alargara excesivamente y hubiera que invertir más en RRHH. Suceda una cosa o la otra, el imprevisto queda cubierto por el presupuesto de contingencia mostrado en el presupuesto final.

Capítulo 11

Informe de Sostenibilidad

Este capítulo está dedicado a reflexionar sobre la sostenibilidad del proyecto. Con sostenibilidad nos referimos a la capacidad de preservar el equilibrio entre economía, ambiente y sociedad; es decir, ofrecer progreso pero sin consumir recursos excesivamente. En las tres secciones que siguen vamos a reflexionar sobre del trabajo en sus 3 fases de desarrollo, teniendo en cuenta las dimensiones ambientales, económicas y sociales.

11.1. Proyecto Puesto en Producción (PPP)

Incluye la planificación, el desarrollo y la implantación del proyecto.

11.1.1. Dimensión Ambiental

El único impacto ambiental que ha supuesto este TFG recae sobre el alquiler de la oficina y el uso del material informático. No es posible cuantificar lo que ha consumido el alquiler de la oficina, pues ésta es compartida con más trabajadores con hábitos muy diferentes y porque la información la gestiona exclusivamente el departamento de contabilidad del BSC. Respecto al material informático, no se ha cuantificado su impacto ambiental, el cual está asociado al consumo de electricidad a lo largo del proyecto. Aún así, dado que los clústers HPC usados son compartidos con más usuarios y que el portátil personal será reciclado o reutilizado, estimamos que el impacto ambiental se mantiene dentro de los valores establecidos por el BSC.

El recurso más consumido ha sido siempre la electricidad porque es lo que alimenta a los supercomputadores, y nosotros hemos dependido casi exclusivamente de ellos. Por lo tanto, para reducir el impacto ambiental nos hemos centrado en minimizar su uso mediante un diseño preciso de los experimentos, para así evitar repetirlos. La reducción que haya supuesto esto no ha sido cuantificada, aún así, si tuviéramos la oportunidad de realizar el proyecto de nuevo reduciríamos considerablemente el consumo energético de los clústers HPC porque a estas alturas conocemos la configuración óptima del software utilizado.

11.1.2. Dimensión Económica

El coste de los recursos humanos y materiales ha sido calculado, a excepción de los clústers HPC que se justificó en el Capítulo 10. Consideramos que las decisiones tomadas durante la planificación inicial del proyecto no se pueden mejorar para reducir los costes de manera considerable. Lo que sí es verdad, es que el imprevisto ocurrido con la máquina CTE-KNL ha supuesto un gasto que no se ha podido amortizar de ninguna manera, pero nosotros no podíamos preverlo. Solamente podríamos considerar el posible riesgo que tal suceso ocurriera eventualmente y añadirlo al presupuesto de contingencia.

Dado que la planificación inicial se ha ajustado bastante bien al desarrollo del proyecto, a pesar de las desviaciones e imprevistos ocurridos, el coste final del proyecto se ha ajustado al presupuesto inicial. De todos modos, el coste extra de las desviaciones e imprevistos ha quedado cubierto por el presupuesto de contingencia.

11.1.3. Dimensión Social

La neutralidad de este estudio no ha implicado ninguna decisión ética o reflexión social. De hecho, ha motivado a la colaboración interdepartamental (departamento de Computer Science y CASE) del BSC y ha estrechado lazos entre sus trabajadores. A nivel personal, ha servido para desarrollar nuestras competencias profesionales.

11.2. Vida Útil

Empieza una vez implantado el proyecto y acaba con su desmantelamiento.

11.2.1. Dimensión Ambiental

Los contenedores son software, por lo tanto, consumirán 0 recursos materiales. Los resultados de este proyecto tampoco van a consumir recursos humanos, pues es solamente un estudio con un inicio y final definido. Sí que es verdad que requerirán de electricidad para poder ser ejecutados en un ordenador, pero este consumo será insignificante porque no es un programa que requiera potencia de cálculo. Además, hemos demostrado que el uso de contenedores puede igualar rendimientos nativos, así que su uso tampoco empeorará el consumo energético de las aplicaciones científicas.

Globalmente, este trabajo motivará el uso de los contenedores mejorando la portabilidad de las aplicaciones. Esta mejora supone directamente invertir menos horas de CPU en los supercomputadores intentando instalar aplicaciones y resolviendo conflictos y aumentando la eficiencia de los clústers HPC. En consecuencia, este proyecto tiene el poder de disminuir la huella ecológica.

11.2.2. Dimensión Económica

Como las implementaciones de contenedores evaluadas son gratuitas, su uso no supone ningún coste económico para nosotros. Es cierto que en el caso de Docker o Singularity existe la versión *enterprise* del software, pero esta versión no es imprescindible y queda a elección de los interesados adquirirla.

11.2.3. Dimensión Social

Tal como está especificado en la Sección de *Agentes Implicados* 1.4 del Capítulo 1, los beneficiaros de este estudio son toda la comunidad HPC, desde los científicos hasta los mánagers de instalaciones HPC.

Los resultados que exponemos podrían perjudicar el uso de Docker y Shifter en HPC porque declaramos Singularity como la mejor implementación actualmente. En el caso de Docker, ésto no supone un impacto muy negativo, ya que su nicho de mercado nunca ha sido el sector del *High-Performance Computing*. A Shifter sí que le afecta, pues es la competencia directa de Singularity. Aún así, como Shifter es un proyecto sin ánimo de lucro y está más enfocado a la investigación de los contenedores, no pensamos que nuestro estudio le perjudique.

El problema que atacamos es la falta de portabilidad de las aplicaciones y la desconfianza del uso de los contenedores en HPC. Con nuestras conclusiones hemos determinado que esta tecnología es adecuada para HPC y por lo tanto ofrece unos argumentos a favor muy sólidos para fomentar su uso. En el futuro, los centros HPC que duden si ofrecer contenedores a sus usuarios, o no, podrán referirse a este trabajo. Es nuestra intención que el BSC siga nuestras recomendaciones respecto a este tópico.

11.3. Riesgos

Los riesgos inherentes al propio proyecto durante toda su ejecución, vida útil y desmantelamiento.

11.3.1. Dimensión Ambiental

No hay riesgos de que el uso de los contenedores empeore la huella ecológica.

11.3.2. Dimensión Económica

Existía el riesgo de tener que adquirir un clúster en caso que Lenox no sirviera para realizar la comparativa de contenedores. Esto hubiera supuesto un enorme gasto, tanto económico como ambiental. La parte positiva, es que este hecho tenía unas probabilidades muy bajas de ocurrir. Una vez realizado el estudio de viabilidad este proyecto ha terminado su labor, lo cual no conlleva costes económicos.

11.3.3. Dimensión Social

Impulsar el uso de contenedores no perjudica de ninguna manera a los usuarios. Solamente es otra herramienta a la disposición de los interesados para agilizar su trabajo con las aplicaciones. Aunque apareciera una dependencia de los usuarios con los contenedores, siempre existirá la alternativa de instalar las aplicaciones manualmente como se ha hecho tradicionalmente. Tampoco existe el riesgo que el uso de contenedores conlleve despidos. Es muy extraño que hayan trabajadores solamente dedicados a portar aplicaciones, y en caso que los hubieran, tampoco les debería afectar, puesto que su trabajo evolucionaría a usar esta nueva tecnología.

Capítulo 12

Conclusiones

En este trabajo hemos evaluado la viabilidad del uso de contenedores en entornos HPC. Todos los experimentos se han realizado con el fin de responder a las cuestiones de la Sección 1.3, que tratan algunas de las preocupaciones más relevantes de este ámbito. En orden, nuestra comparativa (Capítulo 6) ha servido para elegir de entre las 3 implementaciones de contenedores más significativas la que mejor se adapta a los requisitos de *High-Performance Computing* (en este caso Singularity) considerando el proceso de instalación, coste de despliegue y rendimiento con aplicaciones reales y *benchmarks*. A continuación, el estudio de portabilidad (Capítulo 7) ha revelado cuán portables son los contenedores y ofrece una vista del rendimiento que pueden obtener los usuarios siguiendo distintos métodos de creación de imágenes. Al final, el test de escalabidad (Capítulo 8) ha servido para demostrar que esta tecnología de virtualización es capaz de exprimir el máximo rendimiento de los supercomputadores con aplicaciones científicas reales. Por lo tanto, el uso de contenedores en HPC es viable, y como implementación ideal sugerimos Singularity.

La ejecución del proyecto ha dado pocos problemas a pesar de las desviaciones e imprevistos ocurridos. La desviación que más ha afectado al trabajo ha sido la de no poder instalar Singularity en el clúster CTE-KNL. En consecuencia, nos hemos visto obligados a prescindir de esta máquina, aunque se justificó en la Sección 9.2 que esto no supone un impacto muy negativo para nuestros objetivos.

Para acabar, de todo lo aprendido consideramos apropiado ofrecer a la comunidad HPC unos consejos. Vamos a enfocarnos en los distintos tipos de usuarios que pueden estar interesados en los contenedores y aportarles las lecciones más útiles.

- **Administradores de sistemas**: Des de el punto de vista de un administrador de sistemas, Docker representa un estándar por su extenso uso en las TI. Sin embargo, en su actual estado de desarrollo, Docker presenta serios problemas de seguridad, complejidad de orquestación y rendimiento, los cuáles son inaceptables en centros HPC. Singularity como competidor totalmente opuesto, está ganando la atención del dominio HPC por su simplicidad, rendimiento y capacidad de integración con entornos MPI y SLURM. Aparte, también hemos estudiado otras soluciones emergentes como Shifter, que sigue la misma idea que Singularity. A pesar de todo, no sugerimos el uso de Shifter como una solución general porque claramente le falta madurez, tanto en rendimiento como en funcionalida-

des, simplicidad y documentación.

- **Mánagers de instalaciones**: En este trabajo hemos considerado los contenedores como solución para mejorar la portabilidad de las aplicaciones y así mitigar la divergencia de sistemas HPC del mercado. Hemos testeado 3 clústers con 3 arquitecturas distintas representando el estado del arte de los sistemas HPC: la x86 con procesadores Intel Platinum, la Arm con Cavium ThunderX y la PowerPC con IBM Power9. En estos tests hemos evaluado el rendimiento de un caso de uso real cuyos hallazgos pueden ayudar a los mánagers de instalaciones a planificar qué servicios ofrecer a sus usuarios. Con contenedores es posible montar una imagen totalmente independiente del entorno de su anfitrión, aunque esto no asegura el máximo rendimiento. Como solución a esto hemos propuesto que los mánagers publiquen imágenes de contenedores ya preparadas para soportar las características de cada sistema, y así habilitar para los usuarios una manera fácil de crear imágenes optimizadas. Por último, pero no menos importante, hemos evaluado el coste de despliegue de aplicaciones usando contenedores para cuantificar el tiempo que se invierte en preparar el entorno de ejecución. Esto tiene un impacto directo en la eficiencia global de las instalaciones HPC, y suponemos que un buen mánager siempre querrá minimizar este coste. Por eso, una vez más, sugerimos Singularity ya que el coste de Docker es inadmisible. Somos conscientes que nuestro estudio se centra en el rendimiento de la CPU y la red, y que carece de evaluaciones de entrada/salida, almacenamiento de datos distribuidos y GPUs, lo cual podría ser un interesante trabajo futuro.

- **Expertos HPC**: La comparativa de rendimiento de las 3 implementaciones de contenedores han mostrado que Singularity y Shifter ofrecen rendimientos equivalentes al entorno nativo hasta 112 procesos MPI. Docker, por el contrario, sufre una degradación de rendimiento debido a la comunicación mediante la red. Un experto HPC debería tener en cuenta estas observaciones para maximizar los rendimientos de las aplicaciones al usar contenedores. Con el test de escalabilidad hemos verificado que la tecnología de contenedores puede escalar al mismo ratio que las ejecuciones nativas. Aún así, los expertos quedan advertidos de que una buena escalabilidad se obtiene sacrificando la portabilidad de los contenedores.

- **Científicos**: Los últimos comentarios van dedicados a los científicos interesados más en obtener los resultados de las ejecuciones que no en el rendimiento. Resulta interesante saber que, en el caso de códigos para HPC, es posible discernir cuál es la mejor implementación de contenedor para cada escenario. Particularmente, un científico puede estar interesado en usar entornos en la nube y HPC al mismo tiempo. Por ejemplo, puede ser mucho mejor correr simulaciones pequeñas en la nube y las más pesadas en HPC. Entonces, Docker resulta una solución mucho más flexible para casos simples y rápidos, mientras que Singularity se usaría con los casos más lentos y que requieren más recursos. Combinar inteligentemente ambas opciones ayudaría a los científicos a conseguir soluciones aptas tanto para HPC como para la nube, sin la necesidad de sacrificar rendimiento en ningún caso.

Por lo personal, este trabajo ha sido una oportunidad excepcional para entrar en contacto con el mundo del *High-Performance Computing* y la investigación. He tenido la oportunidad de trabajar con varios clústers y supercomputadores de todo tipo de arquitecturas, utilizar *benchmarks* HPC, trabajar conjuntamente con físicos e ingenieros de diversas especialidades, utilizar una aplicación científica real para simular un caso de uso biológico en producción y familiari-

zarme con los modelos de programación paralelos en el ámbito profesional. Además, las conclusiones de los experimentos son muy positivas, por lo que pueden resultar muy útiles en la literatura de este campo y ayudar a los centros HPC a optimizar sus herramientas de trabajo. De hecho, este TFG ya forma parte del estado del arte de la evaluación de la tecnología de contenedores en HPC porque parte de sus resultados han sido presentados en el 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS) bajo el título *Containers in HPC: A Scalability and Portability Study in Production Biological Simulations* [32]. A día de hoy, este paper está pendiente de ser publicado oficialmente en los *proceedings* del año 2019 de esta conferencia. Por si fuera poco, todos los hallazgos de este proyecto forman, o van a formar parte, de los informes técnicos presentados a la Comisión Europea vía HPC-Europa3 para estimular el uso de los contenedores en los centros HPC.

Bibliografía

[1] Jason Nieh y Ozgur Can Leonard. "Examining VMware". En: *Dr. Dobb's Journal* 25.8 (2000).

[2] Peng Li. "Selecting and using virtualization solutions: our experiences with VMware and VirtualBox". En: *Journal of Computing Sciences in Colleges* 25.3 (2010), págs. 11-17.

[3] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." En: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, pág. 46.

[4] Daniel Bartholomew. "Qemu: a multihost, multitarget emulator". En: *Linux Journal* 2006.145 (2006), pág. 3.

[5] N. G. Bachiega y col. "Container-Based Performance Evaluation: A Survey and Challenges". En: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. Abr. de 2018, págs. 398-403. DOI: 10.1109/IC2E.2018.00075.

[6] À. Kovàcs. "Comparison of different Linux containers". En: *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*. Jul. de 2017, págs. 47-51. DOI: 10.1109/TSP.2017.8075934.

[7] Charles Anderson. "Docker [software engineering]". En: *IEEE Software* 32.3 (2015).

[8] Gregory M Kurtzer, Vanessa Sochat y Michael W Bauer. "Singularity: Scientific containers for mobility of compute". En: *PloS one* 12.5 (2017).

[9] A. Kivity y col. "kvm: the Linux Virtual Machine Monitor". En: *Proceedings of the Linux Symposium*. Vol. 1. Ottawa, Ontario, 2007, págs. 225-230.

[10] Jack S Hale y col. "Containers for Portable, Productive, and Performant Scientific Computing". En: *Computing in Science & Engineering* 19.6 (2017), págs. 40-50.

[11] Andrew J Younge y col. "A Tale of Two Systems: Using Containers to Deploy HPC Applications on Supercomputers and Clouds". En: *Cloud Computing Technology and Science (CloudCom), 2017 IEEE Int. Conference on*. IEEE. 2017, págs. 74-81.

[12] Douglas M Jacobsen y Richard Shane Canon. "Contain this, Unleashing Docker for HPC". En: *Proceedings of the Cray User Group* (2015).

[13] Cristian Ruiz, Emmanuel Jeanvoine y Lucas Nussbaum. "Performance evaluation of containers for HPC". En: *European Conference on Parallel Processing*. Springer. 2015, págs. 813-824.

[14] Lucas Benedicic y col. "Portable, high-performance containers for HPC". En: *arXiv preprint arXiv:1704.03383* (2017).

[15] Animesh Kuity y Sateesh Kumar Peddoju. "Performance Evaluation of Container-Based High Performance Computing Ecosystem Using OpenPOWER". En: *International Conference on High Performance Computing*. Springer. 2017, págs. 290-308.

[16] Brian D O'Connor y col. "The Dockstore: enabling modular, community-focused sharing of Docker-based genomics tools and workflows". En: *F1000Research* 6 (2017).

[17] Maxim Belkin y col. "Container Solutions for HPC Systems: A Case Study of Using Shifter on Blue Waters". En: *Proceedings of the Practice and Experience on Advanced Research Computing*. PEARC '18. Pittsburgh, PA, USA, 2018, 43:1-43:8. ISBN: 978-1-4503-6446-1. DOI: 10.1145/3219104.3219145.

[18] Reid Priedhorsky y Tim Randles. "Charliecloud: Unprivileged Containers for User-defined Software Stacks in HPC". En: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017, 36:1-36:10. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126925. URL: http://doi.acm.org/10.1145/3126908.3126925.

[19] Brandon Barker. "Message passing interface (mpi)". En: *Workshop: High Performance Computing on Stampede*. Vol. 262. 2015.

[20] Maximilien de Bayser y Renato Cerqueira. "Integrating MPI with Docker for HPC". En: *Cloud Engineering (IC2E), 2017 IEEE International Conference on*. IEEE. 2017, págs. 259-265.

[21] N. Rajovic, A. Rico, F. Mantovani y col. "The Mont-Blanc Prototype: An Alternative Approach for HPC Systems". En: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. de 2016, págs. 444-455. DOI: 10.1109/SC.2016.37.

[22] Leonardo Dagum y Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming". En: *Computing in Science & Engineering* 1 (1998), págs. 46-55.

[23] Eva Casoni y col. "Alya: computational solid mechanics for supercomputers". En: *Archives of Computational Methods in Engineering* 22.4 (2015), págs. 557-576.

[24] Guillaume Houzeaux y col. "A massively parallel fractional step solver for incompressible flows". En: *Journal of Computational Physics* 228.17 (2009), págs. 6316-6332.

[25] Mariano Vázquez y col. "Alya: towards exascale for engineering simulation codes". En: *arXiv preprint arXiv:1404.4881* (2014).

[26] Ulrich Küttler y Wolfgang A. Wall. "Fixed-point fluid–structure interaction solvers with dynamic relaxation". En: *Computational Mechanics* 43.1 (dic. de 2008), págs. 61-72. ISSN:

1432-0924. DOI: 10.1007/s00466-008-0255-5. URL: https://doi.org/10.1007/s00466-008-0255-5.

[27] Alfonso Santiago y col. "Fully coupled fluid-electro-mechanical model of the human heart for supercomputers". En: *Int. journal for numerical methods in biomedical engineering* (2018).

[28] Ted Belytschko y col. *Nonlinear Finite Elements for Continua and Structures*. John Wiley & sons, 2013.

[29] Rob F vanderWijngaart y Jin Haopiang. "Nas Parallel Benchmarks, Multi-Zone Versions". En: *Supercomputing* (2003).

[30] J Dongarra, MA Heroux y P Luszczek. "HPCG benchmark: a new metric for ranking high performance computing systems. University of Tennessee". En: *Electrical Engineering and Computer Sciente Department, Technical Report UT-EECS-15-736* (2015).

[31] Niall Wilson, Oleksandr Rudyy y Atte Sillanpää. *HPC-Europa3 - D12.2 - Container-as-a-service Technical Documentation*. Project Deliverable. Irish Centre for High End Computing (ICHEC), Abr de 2019. DOI: 10.23728/b2share.2510157e0a4144559683ba87da191e2a.

[32] O. Rudyy y col. "Containers in HPC: A Scalability and Portability Study in Production Biological Simulations". En: *IPDPS2019: Proceedings of the International Parallel and Distributed Processing Symposium*. 2019.